Machine Learning

Chapter 10. Learning Sets of Rules

Vani Rajasekar

Learning set of rules

- One of the most expressive and human readable representations for learned hypotheses is set of if-then rules
- Algorithms
  - Learning set of rules containing variables called first – order rules or horn clauses
  - This can be implemented using PROLOG and is called inductive logic programming (ILP)

Key Aspects

- If-then rules
  - Decision trees
  - Genetic algorithms
- Key Aspects
  - First-order Rules are more expressive than propositional rules
  - Sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules

First – order rule sets

- Consider the following two rules that jointly describe the target concept **Ancestor**
- Here we use the predicate Parent(x,y) to indicate that y is the mother or father of x
- Predicate Ancestor(x,y) to indicate that y is an ancestor of x
- Two rules

  IF Parent(x,y) THEN Ancestor(x,y)

  IF Parent(x,z) and Ancestor(z,y) THEN Ancestor(x,y)
- These two rules describe recursive function that is difficult to represent using decision trees

First order rules applications

- To learn which chemical bonds fragment in a mass spectrometer
- To learn which chemical substructures produce mutagenic activity
- To learn to design finite element meshes to analyze stresses in physical structures

1. Learning Disjunctive Sets of Rules

- Method 1: Learn decision tree, convert to rules
- Method 2: Sequential covering algorithm:

1. Learn one rule with high accuracy, any coverage
2. Remove positive examples covered by this rule
3. Repeat

This algorithm is for learning set of propositional rules(variables-free)

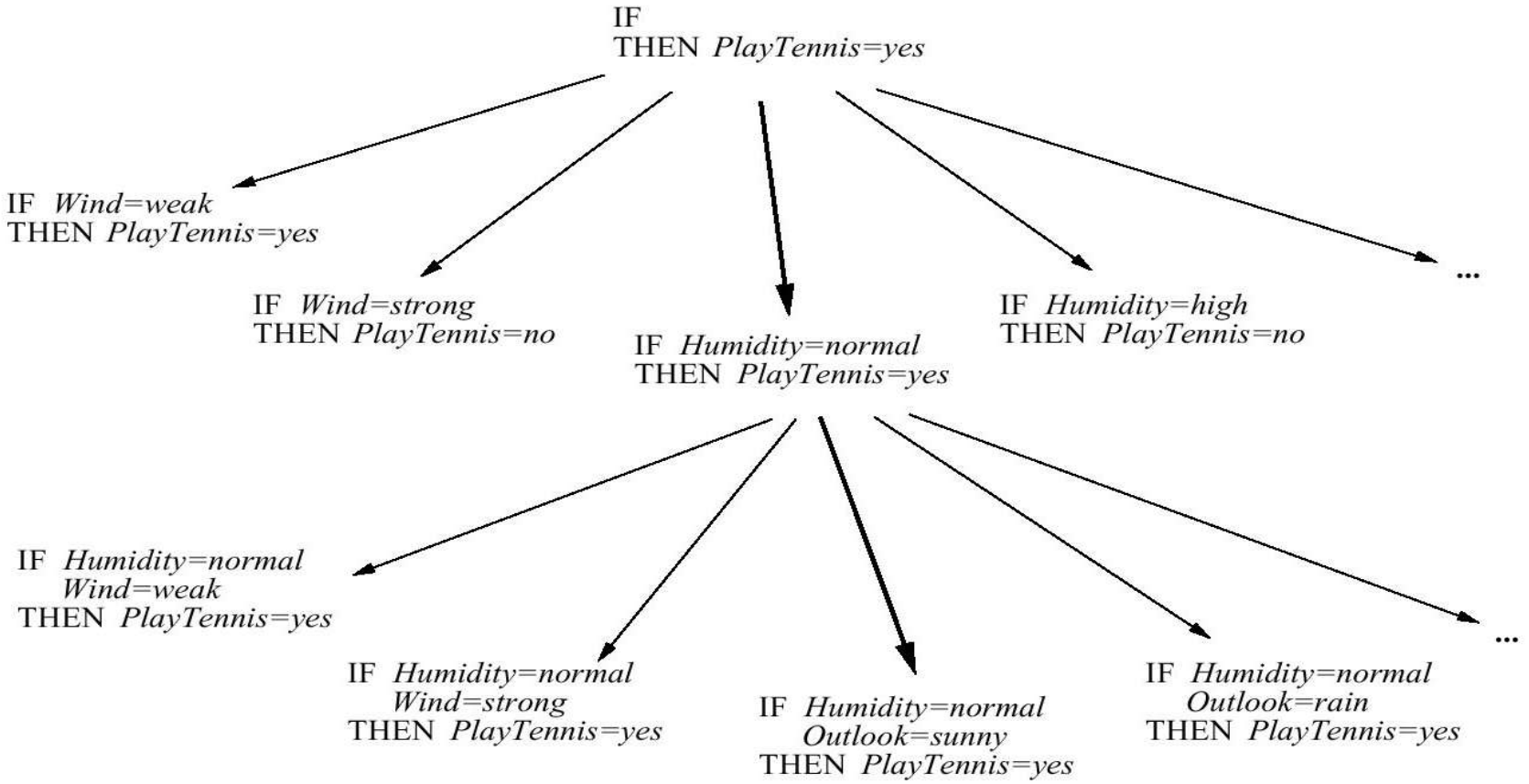
Sequential Covering Algorithm for learning disjunctive set of rules

SEQUENTIAL-COVERING (*Target attribute; Attributes; Examples; Threshold*)

- Learned rules ← {}
- Rule ← LEARN-ONE-RULE(*Target_attribute, Attributes, Examples*)
- while PERFORMANCE (*Rule, Examples*) > *Threshold*, do
  - Learned_rules ← Learned_rules + *Rule*
  - *Examples* ← *Examples* – {examples correctly classified by *Rule*}
  - *Rule* ← LEARN-ONE-RULE (*Target_attribute, Attributes, Examples*)
- *Learned_rules* ← sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*
- Here PERFORMANCE routine is used to evaluate rule quality

2. Learn-One-Rule

The search for rule preconditions as Learn-One-Rule proceeds from general to specific. At each step the preconditions of the best rule are specialized. This is greedy depth first search with no backtracking. The drawback is that a suboptimal choice will be made at any step. To reduce this risk, we can extend the algorithm to perform beam search. The algorithm maintains a list of k best candidates at each step rather than single best candidate.



Implementation of Learn-One-Rule(General – to-specific beam search)

LEARN-ONE-RULE(*Target_attribute, Attributes, Examples, k*)

Returns a single rule that covers some of the Examples. Conducts a general_to_specific greedy beam search for the best rule, guided by the PERFORMANCE *metric.*

- Initialize *Best_hypothesis* to the most general hypothesis $\emptyset$
- Initialize *Candidate_hypotheses* to the set {*Best_hypothesis*}
- While *Candidate_hypotheses* is not empty, Do
  1. *Generate the next more specific candidate_hypotheses*
     - *All_constraints* ← the set of all constraints of the form $(a = v)$, where a is a member of *Attributes*, and v is a value of a that occurs in the current set of *Examples*
     - *New_candidate_hypotheses* ←
       for each h in *Candidate_hypotheses*,
       for each c in *All_constraints*,
       - create a specialization of h by adding the constraint c
     - Remove from *New_candidate_hypotheses* any hypotheses that are duplicates, inconsistent, or not maximally specific
  2. *Update Best_hypothesis*
     - For all h in *New_candidate_hypotheses* do
       - If (PERFORMANCE(h, *Examples*, *Target_attribute*)
         > PERFORMANCE(*Best_hypothesis*, *Examples*, *Target_attribute*))
         Then *Best_hypothesis* ← h
  3. *Update Candidate_hypotheses*
     - *Candidate_hypotheses* ← the k best members of *New_candidate_hypotheses*, according to the PERFORMANCE measure.
- Return a rule of the form
  "IF *Best_hypothesis* THEN *prediction*"
  where *prediction* is the most frequent value of *Target_attribute* among those *Examples* that match *Best_hypothesis*.

PERFORMANCE(h, *Examples*, *Target_attribute*)

- *h_examples* ← the subset of *Examples* that match h
- return $-Entropy(h\_examples)$, where entropy is with respect to *Target_attribute*

Common Evaluation Functions

- Relative Frequency $\frac{n_c}{n}$
  - where n_c = correct rule predictions, n = all predictions
- M-estimate of accuracy $\frac{n_c + mp}{n + m}$
  - Where p = prior probability and m=weight
- Entropy
  - $E(S) = \Sigma^{c}_{i=1} -p_i \log_2 p_i$,

3. Learning First-Order Rules

- This algorithm is for learning set of rules that contain variables
- First-order Horn Clauses
- Terminologies
  - All expressions are composed of
    - constants (Bob),
    - variables (x,y..),
    - predicate symbols(married, >),
    - function symbols(age)
  - Predicate takes the values true or false
  - Function takes any constant as value
  - Term (constant, variable or any function)
  - Literal (any predicate applied to any set of terms-negative literal, positive literal)

First-Order Horn Clauses

To see the advantages of first-order representations over propositional (variable-free) representations, consider the task of learning the simple target concept *Daughter*(x, y), defined over pairs of people x and y. The value of *Daughter*(x, y) is *True* when x is the daughter of y, and *False* otherwise. Suppose each person in the data is described by the attributes *Name*, *Mother*, *Father*, *Male*, *Female*. Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute *Daughter*. For example, the following is a positive example in which Sharon is the daughter of Bob:

$$\begin{array}{lll} \langle Name_1 = Sharon, & Mother_1 = Louise, & Father_1 = Bob, \\ Male_1 = False, & Female_1 = True, & \\ Name_2 = Bob, & Mother_2 = Nora, & Father_2 = Victor, \\ Male_2 = True, & Female_2 = False, & Daughter_{1,2} = True \rangle \end{array}$$

IF $(Father_1 = Bob) \wedge (Name_2 = Bob) \wedge (Female_1 = True)$
THEN $Daughter_{1,2} = True$

Although it is correct, this rule is so specific that it will rarely, if ever, be useful in classifying future pairs of people. The problem is that propositional representations offer no general way to describe the essential *relations* among the values of the attributes. In contrast, a program using first-order representations could learn the following general rule:

IF $Father(y, x) \wedge Female(y)$, THEN $Daughter(x, y)$

where x and y are variables that can be bound to any person.

Basic Definitions from First-Order Logic

- Every well-formed expression is composed of *constants* (e.g., $Mary$, 23, or Joe), *variables* (e.g., x), *predicates* (e.g., $Female$, as in $Female(Mary)$), and *functions* (e.g., age, as in $age(Mary)$).
- A *term* is any constant, any variable, or any function applied to any term. Examples include $Mary$, x, $age(Mary)$, $age(x)$.
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include $Female(Mary)$, $\neg Female(x)$, $Greater\_than(age(Mary), 20)$.
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg Female(Joe)$).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg Female(Joe)$).
- A *positive literal* is a literal with no negation sign (e.g., $Female(Mary)$).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

$$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

Learning First Order Rules

Why do that?

- Can learn sets of rules such as
  Ancestor(x, y) ← *Parent(x; y)*
  Ancestor(x; y) ← *Parent(x; z)* ^ *Ancestor(z; y)*
- General purpose programming language
  PROLOG : programs are sets of such rules

First Order Rule for Classifying Web Pages

[Slattery, 1997]

course(A) ←

has-word(A, instructor),

Not has-word(A, good),

link-from(A, B),

has-word(B, assign),

Not link-from(B, C)

Train: 31/31, Test: 31/34

4. Learning sets of first-order rules(FOIL)

- Extension of sequential covering and learn-one algorithms with two exceptions
  - The rules learned by FOIL are more restricted than general Horn Clauses
  - FOIL rules are more expressive than Horn Clauses
- Outer loop is variant of sequential covering algorithm
- Inner loop corresponds to Learn-One-Rule algorithm

FOIL($Target\_predicate, Predicates, Examples$)

- $Pos \leftarrow$ positive $Examples$
- $Neg \leftarrow$ negative $Examples$
- while Pos, do

  Learn a $NewRule$

  – $NewRule \leftarrow$ most general rule possible

  – $NewRuleNeg \leftarrow Neg$

  – while $NewRuleNeg$, do

  Add a new literal to specialize $NewRule$

  1. $Candidate\_literals \leftarrow$ generate candidates
  2. $Best\_literal \leftarrow$ $\text{argmax}_{L \in Candidate\_literals} Foil\_Gain(L, NewRule)$
  3. add $Best\_literal$ to $NewRule$ preconditions
  4. $NewRuleNeg \leftarrow$ subset of $NewRuleNeg$ that satisfies $NewRule$ preconditions

  – $Learned\_rules \leftarrow Learned\_rules + NewRule$

  – $Pos \leftarrow Pos -$ {members of Pos covered by $NewRule$}

- Return $Learned\_rules$

Information Gain in FOIL

$$Foil\_Gain(L, R) \equiv t \left(\log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)$$

Where

- L is the candidate literal to add to rule R
- p_0 = number of positive bindings of R
- n_0 = number of negative bindings of R
- p_1 = number of positive bindings of $R + L$
- n_1 = number of negative bindings of $R + L$
- t is the number of positive bindings of R also covered by $R + L$

Note

- $-\log_2 \frac{p_0}{p_0+n_0}$ is optimal number of bits to indicate the class of a positive binding covered by R

5. Induction as Inverted Deduction

- A second, quite different approach to inductive logic programming is based on the simple observation that induction is just the inverse of deduction.
- In general, machine learning involves building theories that explain the observed data.
- Given some data D and some partial background knowledge B, learning can be described as generating a hypothesis h that, together with B, explains D.

Induction as Inverted Deduction

- Assume as usual that the training data D is a set of training examples, each of the form $(x_i, f(x_i))$.
- Here x_i denotes the i^{th} training instance and $f(x_i)$ denotes its target value.
- Then learning is the problem of discovering a hypothesis h, such that the classification $f(x_i)$ of each training instance xi follows deductively from the hypothesis h, the description of x_i, and any other background knowledge B known to the system.

Induction as Inverted Deduction

Induction is finding h such that

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ B \wedge h \wedge x_i \vdash f(x_i)$$

where

- x_i is ith training instance
- $f(x_i)$ is the target function value for x_i
- B is other background knowledge

So let's design inductive algorithm by inverting operators for automated deduction!

Induction as Inverted Deduction(Cont')

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i) \qquad (10.2)$$

The expression $X \vdash Y$ is read "Y follows deductively from X," or alternatively "X entails Y." Expression (10.2) describes the constraint that must be satisfied by the learned hypothesis h; namely, for every training instance x_i, the target classification $f(x_i)$ must follow deductively from B, h, and x_i.

Induction as Inverted Deduction

- The mentioned Expression describes the constraint that must be satisfied by the learned hypothesis h; namely, for every training instance x_i, the target classification $f(x_i)$ must follow deductively from B, h, and x_i.

Induction as Inverted Deduction(Cont')

"pairs of people, $\langle u, v \rangle$ such that child of u is v,"

$$f(x_i): \quad Child(Bob, Sharon)$$
$$x_i: \quad Male(Bob), Female(Sharon), Father(Sharon, Bob)$$
$$B: \quad Parent(u, v) \leftarrow Father(u, v)$$

What satisfies $(\forall \langle x_i, f(x_i) \rangle \in D)\ B \wedge h \wedge x_i \vdash f(x_i)$?

$$h_1: Child(u, v) \leftarrow Father(v, u)$$
$$h_2: Child(u, v) \leftarrow Parent(v, u)$$

Constructive Induction

Note that the target literal $Child(Bob, Sharon)$ is entailed by $h_1 \wedge x_i$ with no need for the background information B. In the case of hypothesis h_2, however, the situation is different. The target $Child(Bob, Sharon)$ follows from $B \wedge h_2 \wedge x_i$, but not from $h_2 \wedge x_i$ alone. This example illustrates the role of background knowledge in expanding the set of acceptable hypotheses for a given set of training data. It also illustrates how new predicates (e.g., $Parent$) can be introduced into hypotheses (e.g., h_2), even when the predicate is not present in the original description of the instance x_i. This process of augmenting the set of predicates, based on background knowledge, is often referred to as *constructive induction.*

Induction as Inverted Deduction(Cont')

Induction is, in fact, the inverse operation of deduction, and cannot be conceived to exist without the corresponding operation, so that the question of relative importance cannot arise. Who thinks of asking whether addition or subtraction is the more important process in arithmetic? But at the same time much difference in difficulty may exist between a direct and inverse operation; … it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction….

(Jevons 1874)

In the remainder of this chapter we will explore this view of induction as the inverse of deduction. The general issue we will be interested in here is designing *inverse entailment operators*. An inverse entailment operator, $O(B, D)$ takes the training data $D = \{\langle x_i, f(x_i)\rangle\}$ and background knowledge B as input and produces as output a hypothesis h satisfying Equation (10.2).

$$O(B, D) = h \text{ such that } (\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

Of course there will, in general, be many different hypotheses h that satisfy $(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$. One common heuristic in ILP for choosing among such hypotheses is to rely on the heuristic known as the Minimum Description Length principle (see Section 6.6).

There are several attractive features to formulating the learning task as finding a hypothesis h that solves the relation $(\forall \langle x_i, f(x_i)\rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$.

Inverse Entailment Operator

We have mechanical *deductive* operators

$F(A, B) = C$, where $A \wedge B \vdash C$

need *inductive* operators

$O(B, D) = h$ where $(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$

Induction as Inverted Deduction(Cont')

Positives:

- Subsumes earlier idea of finding h that "fits" training data
- Domain theory B helps define meaning of "fit" the data

$$B \wedge h \wedge x_i \vdash f(x_i)$$

- Suggests algorithms that search H guided by B

Induction as Inverted Deduction(Cont')

Negatives:

- Doesn't allow for noisy data. Consider

$$(\forall \langle x_i, f(x_i) \rangle \in D)\ (B \wedge h \wedge x_i) \vdash f(x_i)$$

- First order logic gives a *huge* hypothesis space H

  $\rightarrow$ overfitting...

  $\rightarrow$ intractability of calculating all acceptable h's

Inverting Resolution

- General inverse entailment operator constructs hypotheses by inverting a deductive inference rule
- A general method for automated deduction is Resolution rule
- Resolution rule is sound and complete rule for deductive inference in first order logic
- Question
  - Whether we can invert the resolution rule to form an inverse entailment operator

Resolution rule in propositional form

It is easiest to introduce the resolution rule in propositional form, though it is readily extended to first-order representations. Let L be an arbitrary propositional literal, and let P and R be arbitrary propositional clauses. The resolution rule is

$$\frac{\begin{array}{ccc} P & \vee & L \\ \neg L & \vee & R \end{array}}{\begin{array}{ccc} P & \vee & R \end{array}}$$

which should be read as follows: Given the two clauses above the line, conclude the clause below the line. Intuitively, the resolution rule is quite sensible. Given the two assertions $P \vee L$ and $\neg L \vee R$, it is obvious that either L or $\neg L$ must be false. Therefore, either P or R must be true. Thus, the conclusion $P \vee R$ of the resolution rule is intuitively satisfying.

Method for Automated Deduction: Resolution Rule

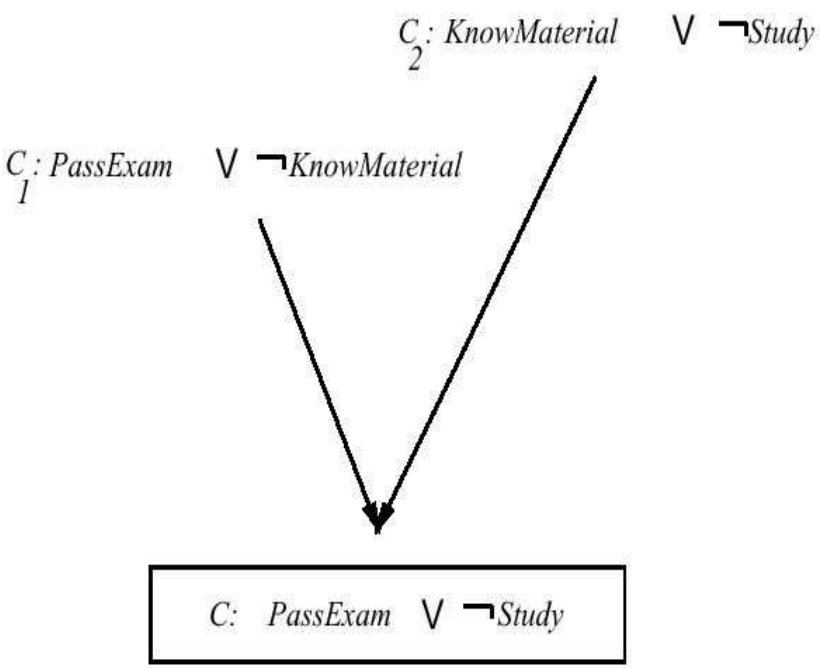
$$\frac{\begin{array}{l} P \vee L \\ \neg L \vee R \end{array}}{P \vee R}$$

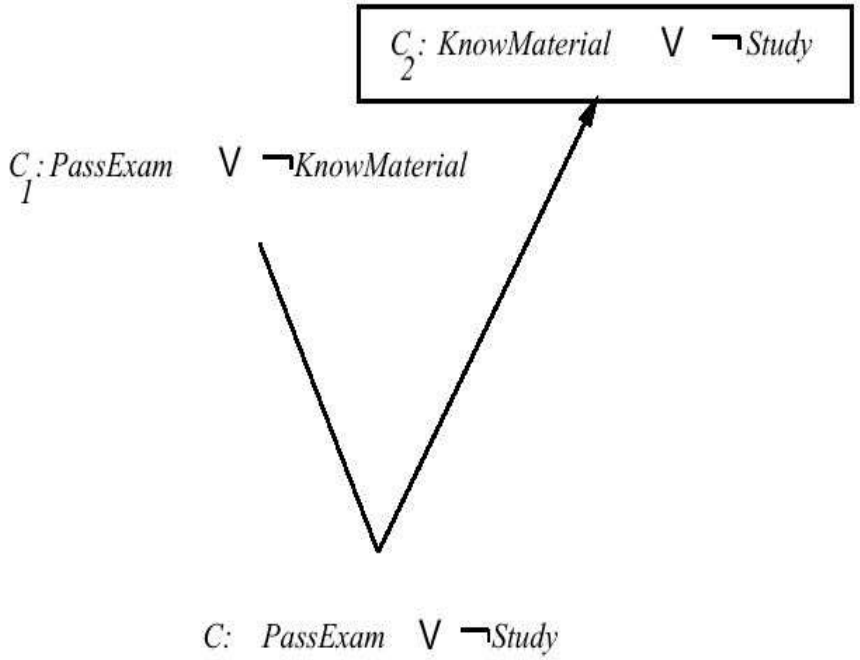
1. Given initial clauses C_1 and C_2, find a literal L from clause C_1 such that $\neg L$ occurs in clause C_2
2. Form the resolvent C by including all literals from C_1 and C_2, except for L and $\neg L$. More precisely, the set of literals occurring in the conclusion C is

$$C = (C_1 - \{L\}) \cup (C_2 - \{\neg L\})$$

where $\cup$ denotes set union, and "$-$" denotes set difference.

Inverting Resolution





Inverted Resolution (Propositional)

1. Given initial clauses C_1 and C, find a literal L that occurs in clause C_1, but not in clause C.
2. Form the second clause C_2 by including the following literals

$$C_2 = (C - (C_1 - \{L\})) \cup \{\neg L\}$$

First Order Resolution

We define a *substitution* to be any mapping of variables to terms. For example, the substitution $\theta = \{x/Bob, y/z\}$ indicates that the variable x is to be replaced by the term Bob, and that the variable y is to be replaced by the term z. We use the notation $W\theta$ to denote the result of applying the substitution θ to some expression W. For example, if L is the literal $Father(x, Bill)$ and θ is the substitution defined above, then $L\theta = Father(Bob, Bill)$.

First order resolution

We say that θ is a *unifying substitution* for two literals L_1 and L_2, provided $L_1\theta = L_2\theta$. For example, if $L_1 = Father(x, y)$, $L_2 = Father(Bill, z)$, and $\theta = \{x/Bill, z/y\}$, then θ is a unifying substitution for L_1 and L_2 because $L_1\theta = L_2\theta = Father(Bill, y)$. The significance of a unifying substitution is this: In the propositional form of resolution, the resolvent of two clauses C_1 and C_2 is found by identifying a literal L that appears in C_1 such that $\neg L$ appears in C_2. In first-order resolution, this generalizes to finding one literal L_1 from clause C_1 and one literal L_2 from C_2, such that some unifying substitution θ can be found for L_1 and $\neg L_2$ (i.e., such that $L_1\theta = \neg L_2\theta$). The resolution rule then constructs the resolvent C according to the equation

$$C = (C_1 - \{L_1\})\theta \cup (C_2 - \{L_2\})\theta \qquad (10.3)$$

Inverting First order resolution

$$C_2 = (C - (C_1 - \{L_1\})\theta_1)\theta_2^{-1} \cup \{\neg L_1 \theta_1 \theta_2^{-1}\}$$

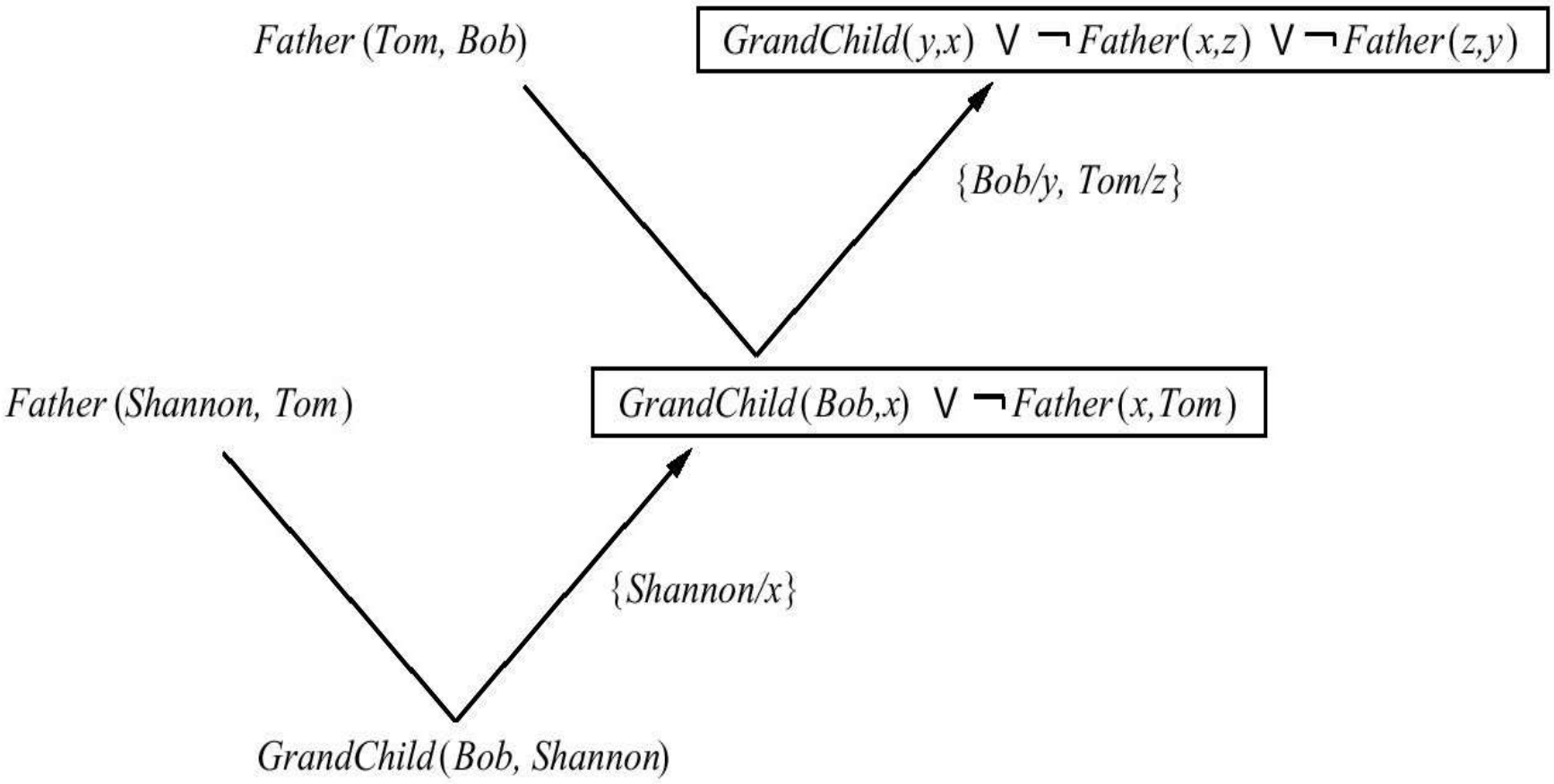
First order resolution

First order resolution:

1. Find a literal L_1 from clause C_1, literal L_2 from clause C_2, and substitution θ such that $L_1\theta = \neg L_2\theta$
2. Form the resolvent C by including all literals from $C_1\theta$ and $C_2\theta$, except for $L_1\theta$ and $\neg L_2\theta$. More precisely, the set of literals occurring in the conclusion C is

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

Cigol


Father (Tom, Bob)
GrandChild(y,x) ∨ ¬Father(x,z) ∨ ¬Father(z,y)
{Bob/y, Tom/z}
Father (Shannon, Tom)
GrandChild(Bob,x) ∨ ¬Father(x,Tom)
{Shannon/x}
GrandChild(Bob, Shannon)

Progol

PROGOL: Reduce comb explosion by generating the most specific acceptable h

1. User specifies H by stating predicates, functions, and forms of arguments allowed for each
2. PROGOL uses sequential covering algorithm. For each $\langle x_i, f(x_i) \rangle$
   - Find most specific hypothesis h_i s.t. $B \wedge h_i \wedge x_i \vdash f(x_i)$
     – actually, considers only k-step entailment
3. Conduct general-to-specific search bounded by specific hypothesis h_i, choosing hypothesis with minimum description length