Logic Programming and Learning

Lecture Schedule

1. Propositional Logic Programming
2. First-order Logic Programming
3. Computations and Answers
4. Introduction to Model Theory
5. Introduction to Proof Theory
6. Generality Orderings
7. Abduction and Justification
8. Search and Redundancy
9. ILP Implementation
10. ILP Experimental Method
11. Revision Class

Symbolic Logic as a computer language

2 stages in software development

1. Specification
   - usually not computer executable
   - correct
2. Implementation
   - computer executable
   - correct
   - efficient

Logic programming is about writing specifications in symbolic logic *and* executing them directly on a computer

Clauses

Statements of the form $p_1 \vee p_2 \ldots \leftarrow q_1 \wedge q_2 \ldots$ are called *clauses*

$p_1 \vee p_2 \ldots$ is sometimes called the *head* of the clause, and $q_1 \wedge q_2 \ldots$ the *body*

If the head has *exactly* 1 proposition without a $\sim$, and the body does not have any $\sim$ symbols, then the clause is called a *definite* clause. Thus:

Clause	Definite clause?
$p \leftarrow q \wedge r$	$\surd$
$p \vee q \leftarrow r \wedge s$	$\times$
$p \leftarrow q \wedge \sim r$	$\times$
$p \leftarrow$	$\surd$

First-order logic: alphabet

Constant symbols. Name specific objects. Start with a lower-case letter ($peter$, $mcmxii$ etc.)

Function symbols. Name a functional relationship between objects. Start with a lower-case letter (sin,cos, $+$ etc.)

Variable symbols. Stand for objects or functions without naming them explicitly. Start with an upper-case letter (X, Y etc.)

Predicate symbols. Name a relation on the world of objects. Start with a lower-case letter (son, $\leq$ etc.)

First-order logic: terms, atoms and quantifiers

Terms

– a constant, variable or functional expression (a function applied to a tuple of terms)

$Expression$	$Term?$
$peter$	$\surd$
X	$\surd$
$log(X)$	$\surd$
$son(peter, peter)$	$\times$
$log(son(peter, peter))$	$\times$
$sin(log(cos(X/2)))$	$\surd$

Atoms

– predicate symbol applied to a tuple of terms ($son(spock, sarek)$)

Arity of function or predicate symbol is the number of terms that each is applied to.

Thus, in $f(a, f(b, Y, Z), q(r(X)))$, the outermost f has arity 1, the inner f has arity 3, q, r have arity 1

- By convention, function and predicate symbols are denoted by $Name/Arity$

Quantifiers

$\forall$ means "for all". It is a way of stating something about all objects in the world without enumerating them. For example, $\forall X\ likes(steve, X)$: $steve$ likes everyone

$\exists$ means "there exists". It is a way of stating the existence of some object in the world without explicitly identifying it. For example, $\exists X\ likes(steve, X)$: $steve$ likes someone

Full Datalog: variables, constants and recursion

Consider the *predecessor* relation, namely, all ordered tuples $< X, Y >\ s.t.\ X$ is an ancestor of Y. This set will include Y's parents, Y's grandparents, Y's grandparents' parents, etc.

$$
\begin{array}{lll}
pred(X,Y) & \leftarrow & parent(X,Y) \\
pred(X,Z) & \leftarrow & parent(X,Y), parent(Y,Z) \\
pred(X,Z) & \leftarrow & parent(X,Y1), parent(Y1,Y2), parent(Y2,Z) \\
\ldots & &
\end{array}
$$

Variables and constants are not enough: we need *recursion*

$\forall X, Z$ X is a predecessor of Z if

1. X is a parent of Z; or
2. X is a parent of some Y, and Y is a predecessor of Z

The *predecessor* relation is thus:

$$
\begin{array}{lll}
pred(X,Y) & \leftarrow & parent(X,Y) \\
pred(X,Z) & \leftarrow & parent(X,Y), pred(Y,Z)
\end{array}
$$

Datalog is not expressive enough

To express arithmetic operations, lists of objects, etc. it is not enough to simply allow variables and constants as terms

– We will need *function* symbols

Consider Peano's postulates for the set of natural numbers $\mathcal{N}$

1. The constant 0 is in $\mathcal{N}$
2. if X is in $\mathcal{N}$ then $s(X)$ is in $\mathcal{N}$
3. There are no other elements in $\mathcal{N}$
4. There is no X in $\mathcal{N}$ $s.t.$ $s(X) = 0$
5. There are no X, Y in $\mathcal{N}$ $s.t.$ $s(X) = s(Y)$ and $X \neq Y$

We can write a definite clause definition for enumerating the elements of $\mathcal{N}$

- 1 constant symbol, 1 unary function symbol

$$natural(0) \leftarrow$$
$$natural(s(X)) \leftarrow natural(X)$$

- They are generated by asking:

natural(N)?

Predicates + Variables + Constants + Functions

Prolog

Computations and answers

Executing definite-clause definitions can sometimes lead to *non-termination* (“infinite loops”) or even *unsound* behaviour (recall the idiosyncratic behaviour of $not/1$)

How are logic programs executed?

1. Execution of propositional logic programs

2. Execution of programs without recursion or negation

3. Execution of programs with recursion but no negation

4. Execution of programs with recursion and negation

Computation and Search rules

Typically, executing a logic program involves solving queries of the form: $l_1, l_2, \ldots, l_n$? where the l_i are literals

Two problems confront us when solving this query:

1. Which literal of the l_i should be solved first?

   – the rule governing this is called the *computation* rule

2. Which clause should be selected first, when more than one can be used to solve the literal selected?

   – the rule governing this is called the *search* rule

Computation and search rules: completeness

Most logic programs are executed using the following:

Computation rule. Leftmost literal first

Search rule. Depth first search for clauses in order of appearance

Question. Will a logic-programming system with an arbitrary computation rule, and a depth-first search of clauses in some fixed order always find a leaf terminating in $SUCCESS$ (if one exists)?

Answer. No

Introduction to model theory

Model theory is concerned with attributing meaning to logical sentences

Basics of model theory

1. Interpretations in propositional logic

2. Model-theoretic notions of validity, logical consequence and satisfiability

3. Interpretations in 1^{st} order logic

4. Herbrand interpretations, Herbrand models for logic programs and minimal Herbrand models

Interpretations: propositional logic

Interpretations are simply assignmnents of $TRUE$ (t) or $FALSE$ (f) to every proposition

- For e.g. given propositions p and q, one possible interpretation assigns p to $TRUE$ and q to $FALSE$

- With this interpretation, other formulae may be true or false: $p \vee q$ is $TRUE$, and $p \wedge q$ is $FALSE$

An interpretation that gives the value $TRUE$ for a formula is called a *model* for that formula

- Thus, $p = TRUE, q = FALSE$ is a model for $p \vee q$

Consequence and equivalence

Consider the formulae p and $p \vee q$

- Every interpretation that makes p true also makes $p \vee q$ true. That is, every model of p is a model of $p \vee q$

If every model of a sentence (or formula) s_1 is also a model of a sentence s_2 then s_2 is said to be a *logical consequence* of s_1. Alternatively, s_1 *logically implies* s_2, or $s_1 \models s_2$

If every model of s_1 is a model of s_2 and every model of s_2 is a model of s_1 then s_1 and s_2 are logically equivalent, or $s_1 \equiv s_2$

Herbrand interpretations and models

Interpretations in 1^{st} order logic are more complex than propositional logic

Yet logic programming systems appear to determine logical consequences without recourse to complex mappings

- Is an "intended interpretation" built-in?
- If so, will it work for any other interpretations?

The logical consequence relation $P \models s$ requires that for *every* interpretation I, if I is a model of P, then it is a model of s

In fact, executing a logic program does not need to consider every interpretation. One special interpretation called the *Herbrand* interpretation is enough

Why?

- A set of clauses P has a model iff P has a Herbrand interpretation that is a model (that is, a "Herbrand model")

- For definite-clause programs, there is a unique minimal Herbrand model

- For any definite-clause program P and ground atom s, $P \models s$ iff s is in the Herbrand model

What are Herbrand interpretations?

Given a program P and a language $\mathcal{L}$ think of all ground terms that can be constructed

– For e.g. let $\mathcal{L}$ consist of the constant symbol 0, functions $s/1, p/1$ and predicate symbol $natural/1$. Let P be:

$$natural(0) \leftarrow$$

$$natural(s(X)) \leftarrow natural(X)$$

– The set of all ground terms that can be constructed is the infinite set $\{0, s(0), p(0), s(p(0)), p(s(0)), \ldots\}$. This set is called the *Herbrand universe*

Now think of all ground atoms that can be constructed using elements from the Herbrand universe and predicate symbols in P

- Here, this is the infinite set $\{natural(0), natural(s(0)), \ldots\}$

- This is called the *Herbrand base* of P or $\mathcal{B}(P)$

A Herbrand interpretation is simply an assignment of $TRUE$ to some subset of $\mathcal{B}(P)$ and $FALSE$ to the rest

- It is common to associate "Herbrand interpretation" only with the set of atoms assigned to $TRUE$

- Thus, $\{natural(0)\}$ is a Herbrand interpretation that assigns $TRUE$ to $natural(0)$ and $FALSE$ to everything else

What are Herbrand models?

Consider the following program P:

$likes(john, X) \leftarrow likes(X, apples)$

$likes(mary, apples) \leftarrow$

- $\mathcal{B}(P)$ is the set: $\{likes(john, john), likes(john, apples), likes(apples, john), likes(john, mary), likes(mary, john), likes(mary, apples), likes(apples, mary), likes(mary, mary), likes(apples, apples)\}$

- $\{likes(mary, apples), likes(john, mary)\}$ is a subset of $\mathcal{B}(P)$, and is a Herbrand interpretation

- It is a Herbrand model for P

- $\{likes(mary, apples), likes(john, mary), likes(mary, john)\}$ is also a model for P

Ground instantiations and Herbrand models

A set of 1^{st} order clauses can be thought of as "short-hand" for a set of ground clauses

- The ground clauses are obtained by replacing variables by terms from the Herbrand universe (i.e. the set of all possible ground terms given $\mathcal{L}$).

- This is called the *ground instantiation* of P or $\mathcal{G}(P)$.

- A program P has a model iff $\mathcal{G}(P)$ has a Herbrand model

Models for definite-clauses

The set of all Herbrand models for a definite-clause program P is partially ordered by $\subseteq$ and forms a lattice. For e.g.

For definite-clause programs, the minimal model is unique

The "meaning" of a definite-clause program is given by its minimal model

Deduction theorem

Let $P = \{s_1, \ldots s_n\}$ be a set of clauses and s be a sentence (not necessarily ground)

Theorem. $P \models s$ iff $P - \{s_i\} \models (s \leftarrow s_i)$

– Implication is preserved if we remove any sentence from the left and make it a condition on the right

$$P - \{s_1, \ldots, s_i\} \models (s \leftarrow s1 \wedge \ldots \wedge s_i)$$

$$\emptyset \models (s \leftarrow s1 \wedge \ldots \wedge s_n)$$

– $s \leftarrow s1 \wedge \ldots \wedge s_n$ is valid

$P \models q$ iff $P \cup \{\sim q\}$ is unsatisfiable

– Logical consequence can be checked by Refutation

Introduction to proof theory

Proof theory considers the mechanics of generating a set of sentences from others

Basics of proof theory

1. Elements of proof theory
2. Theorem proving and proof procedures
3. Resolution for propositional logic
4. Substitutions, and resolution for 1^{st} order logic
5. SLD resolution

Resolution for propositional logic

Consider the clauses:

C_1: $is\_dangerous \leftarrow is\_cheetah$

C_2: $is\_cheetah \leftarrow is\_carnivore, has\_tawny\_colour, has\_dark\_spots$

– The *resolvent* of C_1, C_2 is the clause:

C: $is\_dangerous \leftarrow is\_carnivore, has\_tawny\_colour, has\_dark\_spots$

– Remember

C_1: $is\_dangerous \vee \sim is\_cheetah$

C_2: $is\_cheetah \vee \sim is\_carnivore \vee \sim has\_tawny\_colour \vee \sim has\_dark\_sp$

C: $is\_dangerous \vee \sim is\_carnivore \vee \sim has\_tawny\_colour \vee \sim has\_dark\_spots$

– C_1, C_2 are called the *parent* clauses, and $is\_cheetah$ is the the literal that is resolved upon

Soundness of resolution

A single resolution step does the following:

– From $p \leftarrow q$ and $q \leftarrow r$

– Infer $p \leftarrow r$

Since resolution is sound, we can always add the clauses inferred to the original program

Completeness of resolution

Resolution has these properties

- Consider a set of clauses s.t. each clause has *at most* 1 positive literal. Such clauses are called *Horn* clauses

- If a set of Horn clauses is unsatisfiable then resolution will derive the empty clause. Resolution is thus "refutation complete"

- However, it is not "affirmation complete". That is, if $P \models s$, then it need not follow that $P \vdash s$ using resolution

$$\{p \leftarrow, q \leftarrow\} \models p \leftarrow q$$

- But, if $P \cup \{\sim s\} \;\vdash \Box$ using resolution then $P \cup \{\sim s\} \models \Box$ or $P \models s$

Resolution with 1^{st}-order clauses

Step 0. Given a pair of clauses:

$$C_1: \ likes(steve, X) \ \leftarrow \ buys(X, ilp\_book)$$

$$C_2: \ buys(X, ilp\_book) \ \leftarrow \ sensible(X), rich(X)$$

Step 1. Rename all variables apart.

$$C_1: \ likes(steve, A) \ \leftarrow \ buys(A, ilp\_book)$$

$$C_2: \ buys(B, ilp\_book) \ \leftarrow \ sensible(B), rich(B)$$

Step 2. Identify complementary literals and see if mgu exists.

$$buys(B, ilp\_book)\theta = buys(A, ilp\_book)\theta$$

$$\theta = \{A/B\}$$

Step 3. Apply θ and form resolvent C.

1. Let $C_1\theta = h_1 \vee \sim l_1 \vee \sim l_2 \ldots \vee \sim l_j$
2. Let $C_2\theta = l_1 \vee \sim m_1 \vee \sim m_2 \ldots \vee \sim m_k$

3. Then $C = h_1 \vee \sim m_1 \vee \ldots \vee \sim m_k \vee \sim l_2 \ldots \vee \sim l_j$

Earlier example:

C: $likes(steve, B) \;\leftarrow\; sensible(B), rich(B)$

Resolution remains sound and refutation-complete with clausal logic (proof not required here)

<u>S</u>elected <u>L</u>inear resolution for <u>D</u>efinite clauses

Gven a program P, a query Q $q(\ldots), r(\ldots), \ldots$?

1. Select a literal l_i in Q using some *computation rule*.

2. Select a clause C_i from P that can resolve with the selected literal. If no C_i is possible $FAIL$

3. Construct resolvent C using C_i and $\leftarrow l_i$ as parents

4. If $C = \Box$ $STOP$ otherwise $Q = C$, Goto Step 1

Resolution remains sound and refutation complete with this strategy

Introduction to lattice theory and generality orderings

A lattice is a system of elements with 2 basic operations: formation of meet and formation of join

Basics of lattice theory

1. Sets

2. Relations and operations

3. Equivalence relations

4. Partial orders

5. Lattices

6. Quasi orders

7. Generality orderings

Relevance to ILP

ILP is concerned with the automatic construction of "general" logical statements from "specific" ones.

– For example, given $mem(1, [1, 2]) \leftarrow$ construct $mem(A, [A|B]) \leftarrow$

Questions:

1. What do the words "general" and "specific" mean in a logical setting?

2. Can statements of increasing (decreasing) generality be enumerated in an orderly manner?

These are questions about the mathematics of "generality"

- ILP identifies "generality" with $\models$. That is, C_1 is "more general" than C_2 iff $C_1 \models C_2$

- The relation $\models$ results in a quasi-ordering over a set of clauses.

- ILP systems are programs that search such quasi-ordered sets

Subsumption ordering over atoms

Consider the set S of all atoms in some language, and $S^+ = S \cup \{\top, \bot\}$. Let the dyadic relation $\succeq$ be such that:

- $\top \succeq l$ for all $l \in S^+$

- $l \succeq \bot$ for all $l \in S^+$

- $l \succeq m$ iff there is a substitution θ s.t. $l\theta = m$, for $l, m \in S$

$\succeq$ is a quasi-ordering known as "subsumption". A partial ordering results from the partition of S^+ into the sets $\{[\top]\}, \{[\bot]\}, X_1, \ldots$ where $[l]$ denotes all atoms that are alphabetic variants of l. That is, if $l, m \in X_i$ then there are substitutions μ and σ s.t. $l\mu = m$ and $m\sigma = l$. Thus, $\succeq$ is a partial ordering over the set of equivalence classes of atoms (S_E^+)

Example of subsumption ordering on atoms

- $l = mem(A, [A, B]) \succeq mem(1, [1, 2]) = m$ since with $\theta = \{A/1, B/2\}$, $l\theta = m$
- $mem(A1, [A1, B1]), mem(A2, [A2, B2]) \ldots$ are all members of the same equivalence class

For atoms $l, m \in S$, subsumption is equivalent to implication

- If $l \models m$ then $l \succeq m$

Subsumption lattice of atoms

The p.o. set of equivalence classes of atoms S_E^+ is a lattice with the binary operations $\sqcap$ and $\sqcup$ defined on elements of S_E^+ as follows:

– $[\bot] \sqcap [l] = [\bot]$, and $[\top] \sqcap [l] = [l]$

– If $l_1, l_2 \in S$ have *mgu* θ then $[l_1] \sqcap [l_2] = [l_1\theta] = [l_2\theta]$ otherwise $[l_1] \sqcap [l_2] = [\bot]$

– $[\bot] \sqcup [l] = [l]$, and $[\top] \sqcup l] = [\top]$

– If l_1 and l_2 have *lgg* m then $[l_1] \sqcup [l_2] = [m]$ otherwise $[l_1] \sqcup [l_2] = [\top]$

The join operation or lub called *lgg* stands for least-general-generalisation of atoms

Finite Chains in the Lattice

It can be shown that if $l \succ m$ (l covers m) then there is a finite sequence $l_1, \ldots, l_n$ s.t. $l \succ l_1 \succ \ldots l_n$ where l_n is an alphabetic variant of m

Progress from l_i to l_{i+1} is achieved by applying one of the following substitutions:

1. $\{X/f(X_1, \ldots, X_k)\}$ where X is a variable in l_i, $X_1, \ldots, X_k$ are distinct variables that do not appear in l_i, and f is some k-ary function symbol in the language

2. $\{X/c\}$ where X is a variable in l_i, and c is some constant in the language

3. $\{X/Y\}$ where X, Y are distinct variables in l_i

Subsumption ordering over Horn clauses

Consider the set S of all Horn clauses in some language, and $S^+ = S \cup \{\bot\}$. Let $\Box$ denote the empty clause and the dyadic relation $\succeq$ be such that:

- $\top = \Box \succeq C$ for all $C \in S^+$
- $C \succeq \bot$ for all $C \in S^+$
- $C \succeq D$ iff there is a substitution θ s.t. $C\theta \subseteq D$, for $C, D \in S$

$\succeq$ is a quasi-ordering known as "subsumption". A partial ordering results from the partition of S^+ into the sets $\{[\bot]\}, X_1, \ldots$ where $[C]$ denotes all clauses that are subsume-equivalent to C. This are not simply alphabetic variants (as in the case of atoms).

That is, if $C, D \in X_i$ there are substitutions μ and σ s.t. $C\mu \subseteq D$ and $D\sigma \subseteq C$. In fact, the subsume-equivalent class of C is infinite, and $[C]$ is usually represented by its "smallest" member (*reduced form*). Thus, $\succeq$ is a partial ordering over the set of subsume-equivalent classes of clauses (S_E^+)

Example of subsumption ordering on clauses

- $C = p(X,Y) \leftarrow \succeq p(a,b) \leftarrow q(a,b) = D$ since with $\theta = \{X/a, Y/b\}$, $C\theta \subseteq D$

For clauses $C, D \in S$, subsumption is *not* equivalent to implication

- If $C \succeq D$ then $C \models D$

Subsumption lattice of Horn clauses

The p.o. set of equivalence classes of Horn clauses S_E^+ is a lattice with the binary operations $\sqcap$ and $\sqcup$ defined on elements of S_E^+ as follows:

- $[\bot] \sqcap [C] = [\bot]$, and $[\top] \sqcap [C] = [C]$
- If $C_1, C_2 \in S$ have an *mgi* D then $[C_1] \sqcap [C_2] = [D]$ otherwise $[C_1] \sqcap [C_2] = [\bot]$
- $[\bot] \sqcup [C] = [C]$, and $[\top] \sqcup C] = [\top]$
- If C_1 and C_2 have *lgg* D then $[C_1] \sqcup [C_2] = [D]$ otherwise $[C_1] \sqcup [C_2] = [\top]$

The meet operation or glb called *mgi* stands for most-general-instance. If the set of positive literals in $C_1 \cup C_2$ have an mgu θ, then $mgi(C_1, C_2) = (C_1 \cup C_2)\theta$. Otherwise $mgi(C_1, C_2) = \bot$

The join operation or lub called *lgg* stands for least-general-generalisation of clauses (Lab Nos. 5, 6)

Example

$S^+ \ = \ \{ \ \Box, \ \bot,$

$is\_tiger(tom) \leftarrow has\_stripes(tom), is\_tawny(tom)$,

$is\_tiger(bob) \leftarrow has\_stripes(bob), is\_white(bob)$,

$is\_tiger(tom) \leftarrow has\_stripes(tom)$,

$is\_tiger(tom) \leftarrow is\_tawny(tom)$,

$is\_tiger(bob) \leftarrow has\_stripes(bob)$,

$is\_tiger(bob) \leftarrow is\_white(tom)$,

$is\_tiger(X) \leftarrow has\_stripes(X)$,

$is\_tiger(X) \leftarrow is\_tawny(X)$,

$is\_tiger(X) \leftarrow is\_white(X)$,

$is\_tiger(X) \leftarrow \}$

Diagram of p.o. set S_E^+:

No Finite Chains in the Lattice

The existence of finite chains in lattices of atoms ordered by subsumption does *not* carry over to Horn clauses ordered by subsumption.

This follows from the observation that there are clauses which have no *finite* and complete set of downward covers

Relative Subsumption ordering over Horn clauses

Consider Horn clauses C, D and a set B:

$$D: \quad gfather(henry, john) \leftarrow$$

$$B: \quad \begin{array}{c} father(henry, jane) \leftarrow \\ father(henry, joe) \leftarrow \\ parent(jane, john) \leftarrow \\ parent(joe, robert) \leftarrow \end{array}$$

$$C: \quad gfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$$

Now $C \not\succeq D$. But $C \succeq D'$ where D':

$$\begin{array}{lcl} gfather(henry, john) & \leftarrow & father(henry, jane), \\ & & father(henry, joe), \\ & & parent(jane, john) \\ & & parent(joe, robert) \end{array}$$

Relative subsumption $C \succeq_B D$ if $C \succeq \bot(D, B)$ is a quasi-ordering

- $\bot(B, D)$ may not be Horn
- $\bot(B, D)$ may not be finite

Relative Subsumption Lattice over Horn clauses

Lattice only if B is a finite set of positive ground literals

Least upper bound of Horn clauses C_1, C_2

$$lgg_B(C_1, C_2) = lgg(\perp(B, C_1), \perp(B, C_2))$$

Greatest lower bound of Horn clauses C_1, C_2

$$glb_b(C_1, C_2) = glb(\perp(B, C_1), \perp(B, C_2))$$

The non-existence of finite chains in lattices of Horn clauses ordered by subsumption carries over to the lattice of clauses ordered by relative subsumption

Subsumption ordering over Horn clause-sets

Consider the set S of all finite Horn clause-sets in some language, and $S^+ = S \cup \{\perp\}$. Let $\succeq_\theta$ denote subsumption relation over Horn clauses and the dyadic relation $\succeq$ be such that:

- $\top = \{\Box\} \succeq T$ for all $T \in S^+$
- $T \succeq \perp$ for all $T \in S$
- $T_1 \succeq T_2$ iff $\forall D \in T_2\ \exists C \in T_1$ s.t. $C \succeq_\theta D$

$\succeq$ is a quasi-ordering known as "subsumption". A partial ordering results from the partition of S into the sets $\{\Box\}, X_1, \ldots$ where $[T]$ denotes all clause-sets that are subsume-equivalent to T. Two theories T_1, T_2 are subsume equivalent iff $T_1 \succeq T_2$ and $T_2 \succeq T_1$

Example of subsumption ordering on clause-sets

$$\{mem(A,[A|B]) \leftarrow, mem(A,[B,A|C]) \leftarrow\}$$

$$\succeq$$

$$\{mem(1,[1,2]) \leftarrow, mem(2,[1,2]) \leftarrow\}$$

Subsumption lattice of Horn clause-sets

It can be shown that the p.o. set of equivalence classes of Horn clause-sets S_E^+ is a lattice with the binary operations $\sqcap$ (glb) and $\sqcup$ (lub) defined on elements of S_E^+ (up to subsume-equivalence)

Given a pair $T_1, T_2 \in S_E^+$

$$lub(T_1, T_2) = T_1 \cup T_2$$

Given a pair $T_1, T_2 \in S_E^+$

$$glb(T_1, T_2) = \left\{ gs_{\mathcal{H}}(C_1', C_2') \;\middle|\; \begin{array}{l} \langle C_1, C_2 \rangle \in T_1 \times T_2 \\ \text{and } C_1', C_2' \text{ are variants} \\ \text{of } C_1, C_2 \text{ std. apart} \end{array} \right\}$$

where, using the definition mgi of Horn clauses

$$gs_{\mathcal{H}}(C_1, C_2) = \begin{cases} C_1 \cup C_2 & \text{if } C_1, C_2 \text{ headless} \\ mgi(C_1, C_2) & \text{otherwise} \end{cases}$$

No Finite Chains in the Lattice

The non-existence of finite chains in lattices of Horn clauses ordered by subsumption carries over to Horn clause-sets ordered by subsumption.

The implication ordering

In a manner analogous to subsumption, we can define a quasi-ordering based on implication between clauses (clause-sets)

$$C \succeq D \text{ if } C \models D$$

and a quasi-ordering based on relative implication

$$C \succeq_B D \text{ if } B \cup \{C\} \models D$$

The partial ordering over the resulting equivalence classes is not a lattice (lubs and glbs do not always exist)

Subsumption and Implication

The principal generality orderings of interest are subsumption ($\succeq_\theta$) and implication ($\succeq_\models$)

For clauses C, D, subsumption is *not* equivalent to implication

$$\text{if } C \succeq_\theta D \text{ then } C \succeq_\models D$$

$$\text{but}$$

$$\text{not vice} - \text{versa}$$

For example

$C: \ natural(s(X)) \leftarrow natural(X)$

$D: \ natural(s(s(X))) \leftarrow natural(X)$

The Subsumption Theorem

A key theorem linking subsumption and implication

> If Σ is a set of clauses and D is a clause, then $\Sigma \models D$ iff D is a tautology, or there exists a clause $D' \succeq_\theta D$ which can be derived from Σ using some form of resolution.

When Σ contains a single clause C then the only clauses that can be derived are the result of *self-resolutions* of C

Thus the difference between $C \succeq_{\models} D$ and $C \succeq_\theta D$ arises when C is self-recursive or D is tautological

Tractability

Logical implication between clauses is undecidable (even for Horn clauses)

Subsumption is decidable but NP-complete (even for Horn clauses)

Restrictions to the form of clauses can make subsumption efficient

- Determinate Horn clauses. There exists an ordering of literals in C and exactly one substitution θ s.t. $C\theta \subseteq D$
- $k - local$ Horn clauses. Partition a Horn clause into k "disjoint" sub-parts and perform k independent subsumption tests

More problems with $\models$

We have already looked at the lattice of clauses (quasi-)ordered by subsumption $\succeq_\theta$

The lattice structure implies the existence of $lubs$ (least generalisations) and $glbs$ (greatest specialisations) for pairs of clauses

The same is not true for the implication quasi-ordering $\succeq_{\models}$

Order	lub	glb
$\succeq_\theta$	$\surd$	$\surd$
$\succeq_{\models}$	$\times$	$\surd$

(for restricted languages $lubs$ for $\succeq_{\models}$ may well exist)

Practical Generality Ordering

The strongest quasi-order that is practical appears to be subsumption

Even that will require restrictions on the clauses being compared

Refinement Operators

Refinement operators are defined for a S with a quasi-ordering $\succeq$

- ρ is a *downward refinement operator* if $\forall C \in S : \rho(C) \subseteq \{D | D \in S \text{ and } C \succeq D\}$
- δ is an *upward refinement operator* if $\forall C \in S : \delta(C) \subseteq \{D | D \in S \text{ and } D \succeq C\}$

Desirable properties of ρ (and dually δ)

1. **Locally Finite.** $\forall C \in S$: $\rho(C)$ is finite and computable.
2. **Complete.** $\forall C \succ D$: $\exists E \in \rho^*(C)$ s.t. $E \sim D$
3. **Proper.** $\forall C \in S : \rho(C) \subseteq \{D | D \in S \text{ and } C \succ D\}$

There are no upward (downward) refinement operators that are locally finite, complete and proper for sets of clauses ordered by $\succeq_\theta$

“Inductive” Logic Programming

(Sample data)

Examples:
grandfather(henry,john) $\leftarrow$
grandfather(henry,robert) $\leftarrow$

+

Background:
father(henry,jane) $\leftarrow$
father(henry,joe) $\leftarrow$
parent(jane,john) $\leftarrow$
parent(joe,robert) $\leftarrow$

Hypothesis:
$\forall X, Y$ grandfather(X, Y) $\leftarrow \exists Z$ (father(X, Z), parent(Z, Y))

(A logic program)

Hypothesis formation and justification

Abduction. Process of hypothesis formation.

Justification. The degree of belief assigned to an hypothesis given a certain amount of evidence.

Logical setting for abduction

B	$= C_1 \wedge C_2 \wedge \ldots$	**Background**
E	$= E^+ \wedge E^-$	**Examples**
E^+	$= e_1 \wedge e_2 \wedge \ldots$	Positive Examples
E^-	$= \overline{f_1} \wedge \overline{f_2} \wedge \ldots$	Negative Examples
H	$= D_1 \wedge D_2 \wedge \ldots$	**Hypothesis**

Prior Satisfiability. $B \wedge E^- \not\models \Box$

Posterior Satisfiability. $B \wedge H \wedge E^- \not\models \Box$

Prior Necessity. $B \not\models E^+$

Posterior Sufficiency. $B \wedge H \models E^+$, $B \wedge D_i \models e_1 \vee e_2 \vee \ldots$

More on this later

Probabilistic setting for justification

Bayes' Theorem

$$p(h|E) = \frac{p(h).p(E|h)}{p(E)}$$

Best hypothesis in a set $\mathcal{H}$ (ignoring ties)

$$H \;=\; \mathrm{a}rgmax_{h\in\mathcal{H}}\; p(h|E)$$

Model for Noise Free Data

Given $E = E^{+} \cup E^{-}$

$$p(h|E) \propto D_{\mathcal{H}}(h) \prod_{e \in E^{+}} p(e|h) \prod_{e \in E^{-}} p(e|h)$$

Or

$$P(h|E) \propto D_{\mathcal{H}}(h) \prod_{e \in E^{+}} \frac{D_X(e)}{g(h)} \prod_{e \in E^{-}} \frac{D_X(e)}{1 - g(h)}$$

Noise Free Data (contd.)

Assuming p positive and n negative examples

$$P(h|E) \propto D_{\mathcal{H}}(h) \left(\prod_{e \in E} D_X(e) \right) \left(\frac{1}{g(h)} \right)^p \left(\frac{1}{1 - g(h)} \right)^n$$

Maximal $P(h|E)$ means finding the hypothesis that maximises

$$\log D_{\mathcal{H}}(h) + p \ \log \frac{1}{g(h)} + n \ \log \frac{1}{1 - g(h)}$$

If there are no negative examples, then this becomes

$$\log D_{\mathcal{H}}(h) + p \ \log \frac{1}{g(h)}$$

Hypothesis Formation

Given background knowledge B and positive examples $E^+ = e_1 \wedge e_2 \ldots$, negative examples E^- ILP systems are concerned with finding a hypothesis $H = D_1 \wedge \ldots$ that satisfies (note: $\cup$ and $\wedge$ used interchangeably)

Posterior Sufficiency. $B \wedge H \models E^+$ and
$B \wedge D_j \models e_1 \vee e_2 \vee \ldots$

Posterior Satisfiability. $B \wedge H \wedge E^- \not\models \Box$

Recall that if more than one H satisfies this, the one with highest posterior probability is chosen

The D_i can be found by examining clauses that "relatively subsume" at least one example

Single Example, Single Hypothesis Clause

What does it mean for clause D to "relatively subsume" example e

- Normal subsumption: $D \succeq e$ means $\exists\theta\ s.t.\ D\theta \subseteq e$. This also means $D\theta \models e$ or $\models (e \leftarrow D\theta)$

$$
\begin{array}{rc}
e: & gfather(henry, john) \leftarrow \\
B: & father(henry, jane) \leftarrow \\
 & father(henry, joe) \leftarrow \\
 & parent(jane, john) \leftarrow \\
 & parent(joe, robert) \leftarrow \\
D: & gfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)
\end{array}
$$

- Note that for this B, D, e with $\theta = \{X/henry, Y/john, Z/jane\}$, $B \cup \{D\theta\} \models e$
- That is: $D \succeq_B e$ means $B \models (e \leftarrow D\theta)$ Clearly if $B = \emptyset$ normal subsumption between clauses results.

- Using the Deduction Theorem

$$\begin{aligned} B \models (e \leftarrow D\theta) &\equiv B \cup \{D\theta\} \models e \\ &\equiv B \cup \overline{e} \models \overline{D\theta} \\ &\equiv \{D\theta\} \models \overline{B \cup \overline{e}} \\ &\equiv \models (\overline{B \cup \overline{e}} \leftarrow D\theta) \end{aligned}$$

- That is, $D \succeq_B e$ means $D \succeq \overline{B \cup \overline{e}}$

- Recall that if $C_1 \succeq C_2$ then $C_1 \models C_2$. In fact, if $C_{1,2}$ are not self-recursive, then $C_1 \succeq C_2 \equiv C_1 \models C_2$

- Let $a_1 \wedge a_2 \ldots$ be the ground literals true in all models of $B \cup \overline{e}$. Then

$$B \cup \overline{e} \models a_1 \wedge a_2 \ldots$$
$$\overline{a_1 \wedge a_2 \wedge \ldots} \models \overline{B \cup \overline{e}}$$

- Let $\bot(B, e) = \overline{a_1 \wedge a_2 \wedge \ldots}$.

- if $D \succeq \bot(B, e)$ then $D \models \bot(B, e)$ and therefore $D \models \overline{B \cup \overline{e}}$.

- In fact, it can be shown that if D, e are not self-recursive and $D \succeq \bot(B, e)$ then $D \succeq \overline{B \cup \overline{e}}$ (that is, $D \succeq_B e$)

A Sufficient Implementation (given B, E)

1. $h_0 = B, i = 0, E^+ = \{e_1, \ldots, e_n\}$

2. repeat

   (a) increment i

   (b) Obtain the most specific clause $\bot(B, e_i)$

   (c) Find the clause D_i that: subsumes $\bot(B, e_i)$; and is consistent with the negative examples;

   (d) $h_i = h_{i-1} \cup \{D_i\}$

3. until $i > n$

4. return h_n

- $\perp(B, e_i)$ may be infinite

- May perform a lot of redundant computation ($D_i \in h_{i-1}$)

- Need not return in the hypothesis with maximum posterior probability

A "Greedy" Implementation (given B, E)

1. $h_0 = B, E_0^+ = E^+, i = 0$

2. repeat

   (a) increment i

   (b) Randomly choose a positive example e_i from E_{i-1}^+

   (c) Obtain the most specific clause $\bot(B, e_i)$

   (d) Find the clause D_i that: subsumes $\bot(B, e_i)$; and is consistent with the negative examples; and maximises $p(h_{i-1} \cup \{D_i\} | e_i^+ \cup E^-)$ where e_i^+ are the examples in E^+ made redundant by $h_{i-1} \cup \{D_i\}$

   (e) $h_i = h_{i-1} \cup \{D_i\}$

   (f) $E_i^+ = E_{i-1}^+ \backslash e_i^+$

3. until $E_i^+ = \emptyset$

4. return h_i

- $\perp(B, e_i)$ may be infinite

- Need not return in the hypothesis with maximum posterior probability

Finding $\bot$: an example

B:

gfather(X,Y) $\leftarrow$ father(X,Z), parent(Z,Y)
father(henry,jane) $\leftarrow$
mother(jane,john) $\leftarrow$
mother(jane,alice) $\leftarrow$

e_i**:**

gfather(henry,john) $\leftarrow$

Conjunction of ground atoms provable from $B \cup \overline{e_i}$:

$\neg$parent(jane,john) $\wedge$
father(henry,jane) $\wedge$
mother(jane,john) $\wedge$
mother(jane,alice) $\wedge$
$\neg$gfather(henry,john)

$\bot(B, e_i)$**:**

gfather(henry,john) $\vee$ parent(jane,john) $\leftarrow$
father(henry,jane),
mother(jane,john),
mother(jane,alice)

D_i**:**

parent(X,Y) $\leftarrow$ mother(X,Y)

Ways of obtaining a finite $\bot$: depth-bounded mode language

Finding a clause D_i that subsumes $\bot(B, e_i)$ is hampered by the fact that $\bot(B, e_i)$ may be infinite!

Use constrained subset of definite clauses to construct finite most-specific clauses

Mode declarations and maximum “depth” of variables

Finding $\perp_i$: an example

$\perp(B,\ e_i)$:

gfather(henry,john) $\vee$ parent(jane,john) $\leftarrow$
father(henry,jane),
mother(jane,john),
mother(jane,alice)

modes:

modeh(,parent(+person,-person))*
modeb(,mother(+person,-person))*
modeb(,father(+person,-person))*

$\perp_0(B, e_i)$**:**

parent(X,Y) $\leftarrow$

$\perp_1(B, e_i)$**:**

parent(X,Y) $\leftarrow$
mother(X,Y),
mother(X,Z)

Revised “Greedy” Implementation (given B, E, d)

1. $h_0 = B, E_0^+ = E^+, i = 0$

2. repeat

   (a) increment i

   (b) Randomly choose a positive example e_i from E_{i-1}^+

   (c) Obtain the most specific clause $\bot_d(B, e_i)$

   (d) Find the clause D_i that: subsumes $\bot(B, e_i)$; and is consistent with the negative examples; and maximises $p(h_{i-1} \cup \{D_i\} | e_i^+ \cup E^-)$ where e_i^+ are the examples in E^+ made redundant by $h_{i-1} \cup \{D_i\}$

   (e) $h_i = h_{i-1} \cup \{D_i\}$

   (f) $E_i^+ = E_{i-1}^+ \backslash e_i^+$

3. until $E_i^+ = \emptyset$

4. return h_i

- Need not return in the hypothesis with maximum posterior probability

Search and Redundancy

2 stages in clause-by-clause construction of hypothesis

1. Search

2. Remove redundant clauses once best clause is found

Moving about in the lattice: refinement steps

General-to-specific search: start at $\Box$, and move by

1. Adding a literal drawn from $\bot_i$

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

2. Equating two variables of the same type

   $p(X,Y) \leftarrow q(X)$ becomes $p(X,X) \leftarrow q(X)$

3. Instantiate a variable with a general functional term or constant

   $p(X,Y) \leftarrow q(X)$ becomes $p(3,Y) \leftarrow q(3)$

Specific-to-general search: start at $\bot_i$

Each of these is called a “refinement step”

An Optimal Search Algorithm: Branch-and-Bound

$bb(i, \rho, f)$: Given an initial element i from a discrete set S; a successor function $\rho : S \rightarrow 2^S$; and a cost function $f : S \rightarrow \Re$, return $H \subseteq S$ such that H contains the set of cost-minimal models. That is for all $h_{i,j} \in H, f(h_i) = f(h_j) = f_{min}$ and for all $s' \in S \backslash H \;\; f(s') > f_{min}$.

1. $Active := \langle i \rangle$.
2. $best := \infty$
3. $selected := \emptyset$
4. while $Active \neq \langle\rangle$
5. begin
   (a) remove element k from $Active$
   (b) $cost := f(k)$
   (c) if $cost < best$
   (d) begin
      i. $best := cost$
      ii. $selected := \{k\}$
      iii. let $Prune_1 \subseteq Active$ s.t. for each $j \in Prune_1$, $\underline{f}(j) > best$ where $\underline{f}(j)$ is the lowest cost possible from j or its successors

iv. remove elements of $Prune_1$ from $Active$

(e) end

(f) elseif $cost = best$

i. $selected := selected \cup \{k\}$

(g) $Branch := \rho(k)$

(h) let $Prune_2 \subseteq Branch$ s.t. for each $j \in Prune_2$, $\underline{f}(j) > best$ where $\underline{f}(j)$ is the lowest cost possible from j or its successors

(i) $Bound := Branch \backslash Prune_2$

(j) add elements of $Bound$ to $Active$

6. end

7. return $selected$

Different search methods result from specific implementations of $Active$

– Stack: depth-first search

– Queue: breadth-first search

– Prioritised Queue: best-first search

Redundancy 1: Literal Redundancy

Literal l is redundant in clause $C \vee l$ relative to background B iff

$$B \wedge (C \vee l) \equiv B \wedge C$$

Can show The literal l is redundant in clause $C \vee l$ relative to the background B iff

$$B \wedge (C \vee l) \models C$$

The clause C is said to be reduced with respect to background knowledge B iff no literal in C is redundant.

Redundancy 2: Clause redundancy

Clause C is redundant in the $B \wedge C$ iff $B \wedge C \equiv B$.

Can show Clause C is redundant in $B \wedge C$ iff

$$B \models C \equiv B \wedge \overline{C} \models \Box$$

A set of clauses S is said to be reduced iff no clause in S is redundant

Example

$$e_j: \quad gfather(henry, john) \leftarrow$$

$$\begin{aligned} B: \quad & father(henry, jane) \leftarrow \\ & father(henry, joe) \leftarrow \\ & parent(jane, john) \leftarrow \\ & parent(joe, robert) \leftarrow \end{aligned}$$

$$D_j: \quad gfather(X, Y) \leftarrow father(X, Z), parent(Z, Y)$$

What is meant by "Accuracy"?

Accuracy is measured according to some probability distribution D over the space of possible examples.

> For illustration, examples might be drawn according to the uniform distribution over the Herbrand base for a given set of constants and a given predicate.

The accuracy of a hypothesis H is simply the probability, according to D, of drawing an example that H misclassifies.

Method 1: Get More Examples

We wish to estimate the accuracy of our algorithm's hypothesis H. If we can obtain m additional labelled examples, we can estimate the accuracy of H based on the binomial distribution. Call an example a *success* just if H classifies it correctly, and suppose in m examples we have $n \leq m$ successes. Then $\frac{n}{m}$ is an *unbiased estimator* of the accuracy of H.

Note: if we used the *same* examples for testing as we used for learning, the estimate of accuracy would be biased in an "optimistic way".

Further Details of Method 1

Suppose that p is the true accuracy of our hypothesis H, and we will draw m new examples. Then n (the number of successes) is distributed according to the binomial distribution $b(m, p)$ with mean p and variance $mp(1 - p)$. This additional information allows us to say something about how close our estimate of accuracy is likely to be to the true accuracy.

Problem with Method 1

We often have very limited data: Examples: determining active drugs or protein structure requires time and costly experiments; determining user interests requires time on the part of the user.

So if we had more data for testing, we'd really like to use it for training. But then we lose our unbiased estimator. It turns out we can get tighter, almost unbiased estimates if we do *resampling*. We will consider only one resampling method here.

Method 2: Leave-One-Out Cross-Validation

Suppose we have m examples. We train (learn) using $m-1$ examples, *leaving one out* for testing. We repeat this m times, each time leaving out a different example. The accuracy estimate is the number of successes (correct classifications) divided by m.

Problem with leave-one-out: can be computationally expensive. This motivates our next technique.

Method 2': k-Fold Cross-Validation

k-fold cross-validation is the same as leave-one-out cross-validation, except we only repeat k times, each time training on $m - \frac{m}{k}$ examples and testing on the remaining $\frac{m}{k}$ examples.

Presentation of Results

Results of testing on new data or k-fold cross-validation are tabulated as follows:

		Actual Positive	Actual Negative	
Predicted	Positive	n_1	n_2	n_a
	Negative	n_3	n_4	n_b
		n_c	n_d	m

n_1: number of examples in the test set that are labelled "positive" and are predicted "positive" *etcetera*

$$Accuracy = p = \frac{n_1 + n_4}{m}$$

$$S.d = \sqrt{p(1-p)/m}$$ (not with k-fold c.v.)

Classical statistical tests for independence between "Actual" and "Predicted" values can be applied to the table when testing is done on new data