Aims

This lecture will introduce you to theoretical and applied aspects of representing hypotheses for machine learning in first-order logic. Following it you should be able to:

• outline the key differences between propositional and first-order learning
• describe the problem of learning relations and some applications
• outline the problem of induction in terms of inverse deduction
• describe inverse resolution in propositional and first-order logic
• describe least general generalisation and $\theta$-subsumption
• reproduce the basic FOIL algorithm and its use of information gain

[Recommended reading: Mitchell, Chapter 10]
[Recommended exercises: 10.5 – 10.7 (10.8)]

Acknowledgement: Material derived from slides for the book
http://www-2.cs.cmu.edu/~tom/mlbook.html
and the book Inductive Logic Programming: Techniques and Applications
by N. Lavrac and S. Dzeroski, Ellis Horwood, New York, 1994
(available at http://www-ai.ijs.si/SasoDzeroski/ILPBook/)
and the paper by A. Cootes, S.H. Muggleton, and M.J.E. Sternberg
(available at http://www.doc.ic.ac.uk/~shm/jnl.html)
and the book Logical and Relational Learning, Luc De Raedt,

Relevant programs

Progol
http://www.doc.ic.ac.uk/~shm/progol.html
Aleph
http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph
FOIL
http://www.rulequest.com/Personal/
iProlog
http://www.cse.unsw.edu.au/~claude/research/software/
Golem
http://www.doc.ic.ac.uk/~shm/golem.html
See also:
http://www-ai.ijs.si/~ilpnet2/systems/
Representation in Propositional Logic

Propositional variables: $P, Q, R, \ldots$

Negation: $\neg S, \neg T, \ldots$

Logical connectives: $\land, \lor, \rightarrow, \leftrightarrow$

Well-formed formulae: $P \lor Q, (\neg R \land S) \rightarrow T$, etc.

Inference rules:

- **modus ponens** Given $B$ and $A \leftarrow B$ infer $A$
- **modus tollens** Given $\neg A$ and $A \leftarrow B$ infer $\neg B$

Enable **sound** or valid inference.

Meaning in Propositional Logic

Propositional variables stand for declarative sentences (properties):

- $P$ the paper is red
- $Q$ the solution is acid

Potentially useful inferences:

- $P \rightarrow Q$ If the paper is red then the solution is acid

Meaning of such formulae can be understood with a **truth table**:

<table>
<thead>
<tr>
<th>$P$</th>
<th>$Q$</th>
<th>$P \rightarrow Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Representation in First-Order Predicate Logic

We have a richer language for developing formulae:

- Constant symbols: Fred, Jane, Copper, Manganese, 
- Function symbols: Cons, Succ, 
- Variable symbols: $x, y, z, \ldots$
- Predicate symbols: Parent, Likes, Binds, 

We still have:

- Negation: $\neg$Likes(Bob, Footy), 
- Logical connectives: $\land, \lor, \rightarrow, \leftrightarrow$

but we also have quantification:

- $\forall x$Likes($x$, Fred), $\exists y$Binds(Copper, $y$)

And we still have well-formed formulae and inference rules …

Meaning in First-Order Logic

Same basic idea as propositional logic, but more complicated.

Give meaning to first-order logic formulae by **interpretation** with respect to a given **domain** $D$ by associating:

- each constant symbol with some **element** of $D$
- each $n$-ary function symbol with some **function** from $D^n$ to $D$
- each $n$-ary predicate symbol with some **relation** in $D^n$

For variables, essentially consider associating all or some domain elements in the formula, depending on quantification.

**Interpretation is association of a formula with a truth-valued statement about the domain.**
Learning First Order Rules

Why do that?

- trees, rules so far have allowed only comparisons of a variable with a constant value (e.g., sky = sunny, temperature < 45)
- these are propositional representations – have same expressive power as propositional logic
- to express more powerful concepts, say involving relationships between example objects, propositional representations are insufficient, and we need a more expressive representation

E.g., to classify X depending on it's relation R to another object Y

Learning First Order Rules

BUT in first order logic sets of rules can represent graph concepts such as

\[ \text{Ancestor}(x, y) \leftarrow \text{Parent}(x, y) \]
\[ \text{Ancestor}(x, y) \leftarrow \text{Parent}(x, z) \land \text{Ancestor}(z, y) \]

The declarative programming language Prolog is based on the Horn clause subset of first-order logic – a form of Logic Programming:

- Prolog is a general purpose programming language: logic programs are sets of first order rules
- “pure” Prolog is Turing complete, i.e., can simulate a Universal Turing machine (every computable function)
- learning in this representation is called Inductive Logic Programming (ILP)

PROLOG definitions for relational concepts

Some Prolog syntax:

- all predicate and constant names begin with a lower-case letter
  - predicate (relation) names, e.g. uncle, adjacent
  - constant names, e.g. fred, banana
- all variable names begin with an upper-case letter
  - X, Y, Head, Tail
- a predicate is specified by its name and arity (number of arguments), e.g.
  - male/1 means the predicate “male” with one argument
  - sister/2 means the predicate “sister of” with two arguments
Prolog definitions for relational concepts

- predicates are defined by sets of clauses, each with that predicate in its head
  - e.g. the recursive definition of ancestor/2

  \[
  \text{ancestor}(X,Y) ::= \text{parent}(X,Y).
  \]

  \[
  \text{ancestor}(X,Y) ::= \text{parent}(X,Z), \text{ancestor}(Z,Y).
  \]

- clause head, e.g. ancestor/2, is to the left of the `:-`

- clause body, e.g. parent(X,Z), ancestor(Z,Y), is to the right of the `:-`

- each instance of a relation name in a clause is called a literal
- a definite clause has exactly one literal in the clause head
- a Horn clause has at most one literal in the clause head
- Prolog programs are sets of Horn clauses
- Prolog is a form of logic programming (many approaches)
- related to SQL, functional programming, ...

Induction as Inverted Deduction

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. ... (W.S. Jevons, 1874)
Induction as Inverted Deduction

[From lecture on Concept Learning:] Induction is finding $h$ such that

$$\forall (x_i, f(x_i)) \in D \land h \land x_i \vdash f(x_i)$$

where

- $x_i$ is $i$th 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!

We have mechanical deductive operators $F(A, B) = C$, where $A \land B \vdash C$

need inductive operators

$$O(B, D) = h \text{ where } (\forall (x_i, f(x_i)) \in D) \ (B \land h \land x_i) \vdash f(x_i)$$

Positives:

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

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

- Suggests algorithms that search $H$ guided by $B$
Induction as Inverted Deduction

Negatives:

• Doesn’t allow for noisy data. Consider

\[(\forall (x_i, f(x_i)) \in D) (B \land h \land x_i) \vdash f(x_i)\]

• First order logic gives a huge hypothesis space \(H\)
  \(\rightarrow\) overfitting…
  \(\rightarrow\) intractability of calculating all acceptable \(h\)’s

---

Deduction: Resolution Rule

\[
\frac{P \lor \neg L \lor L \lor R}{P \lor R}
\]

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

Inverting 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

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

3. Given initial clauses \(C_2\) and \(C\), find a literal \(\neg L\) that occurs in clause \(C_2\), but not in clause \(C\).

4. Form the second clause \(C_1\) by including the following literals

\[
C_1 = (C - (C_2 - \{\neg L\})) \cup \{L\}\]
Duce operators

<table>
<thead>
<tr>
<th>Op</th>
<th>Same Head</th>
<th>Different Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Identification</td>
<td>Absorption</td>
</tr>
<tr>
<td></td>
<td>$p \leftarrow A, B$</td>
<td>$p \leftarrow q, B$</td>
</tr>
<tr>
<td></td>
<td>$p \leftarrow A, q$</td>
<td>$q \leftarrow A$</td>
</tr>
<tr>
<td>W</td>
<td>Intra-construction</td>
<td>Inter-construction</td>
</tr>
<tr>
<td></td>
<td>$p \leftarrow A, B_1$</td>
<td>$p_1 \leftarrow w, B_1$</td>
</tr>
<tr>
<td></td>
<td>$p \leftarrow A, B_2$</td>
<td>$p_2 \leftarrow w, B_2$</td>
</tr>
<tr>
<td></td>
<td>$p \leftarrow A, w$</td>
<td>$w \leftarrow A$</td>
</tr>
</tbody>
</table>

Each operator is read as: pre-conditions on left, post-conditions on right.

First order resolution:

1. Find a literal $L_1$ from clause $C_1$, literal $L_2$ from clause $C_2$, and substitution $\theta$ 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

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

Inverting First order resolution

Factor $\theta$

$$C = (C_1 - \{L_1\}) \theta_1 \cup (C_2 - \{L_2\}) \theta_2$$

$C_2$ should have no common literals with $C_1$

$$C - (C_1 - \{L_1\}) \theta_1 = (C_2 - \{L_2\}) \theta_2$$

By definition of resolution $L_2 = \neg L_1 \theta_1 \theta_2^{-1}$

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

Cigol

Father (Tom, Bob) V ¬Father(x,y) V ¬Father(y,z)

{Bob,y, Tom,z}

Father (Shannon, Tom)

GrandChild (Bob, Shannon)
Subsumption and Generality

θ-subsumption  

\[ C \text{-} \theta \text{-subsumes } D \text{ if there is a substitution } \theta \text{ such that } C \theta \subseteq D. \]

\[ C \text{ is at least as general as } D \ (C \leq D) \text{ if } C \text{-} \theta \text{-subsumes } D. \]

If \( C \) \( \theta \)-subsumes \( D \) then \( C \) logically entails \( D \) (but not the reverse).

\( \theta \)-subsumption is a partial order, thus generates a lattice in which any two clauses have a least-upper-bound and a greatest-lower-bound.

The least general generalisation (LGG) of two clauses is their least-upper-bound in the \( \theta \)-subsumption lattice.

LGG


- LGG of clauses is based on LGGs of literals (atoms)
- LGG of literals is based on LGGs of terms, i.e. constants and variables
- LGG of two constants is a variable, i.e. a minimal generalisation

LGG of atoms

Two atoms are compatible if they have the same predicate symbol and arity (number of arguments)

- \( \lgg(a, b) \) for different constants or functions with different function symbols is the variable \( X \)
- \( \lgg(f(a_1, \ldots, a_n), f(b_1, \ldots, b_n)) = f(\lgg(a_1, b_1), \ldots, \lgg(a_n, b_n)) \)
- \( \lgg(Y_1, Y_2) \) for variables \( Y_1, Y_2 \) is the variable \( X \)

Note:

1. must ensure that the same variable appears everywhere its bound arguments do in the atom
2. must ensure introduced variables appear nowhere in the original atoms

LGG of clauses

The LGG of two clauses \( C_1 \) and \( C_2 \) is formed by taking the LGGs of each literal in \( C_1 \) with every literal in \( C_2 \).

Clauses form a subsumption lattice, with LGG as least upper bound and MGI (most general instance) as lower bound.

Leads to relative LGGs with respect to background knowledge.
Subsumption lattice

RLGG – LGG relative to background knowledge

Example from Quinlan (1991)

Given two ground instances of target predicate $Q/k$, $Q(c_1, c_2, \ldots, c_k)$ and $Q(d_1, d_2, \ldots, d_k)$, plus other logical relations representing background knowledge that may be relevant to the target concept, the relative least generalisation (rlgg) of these two instances is:

$$Q(lgg(c_1, d_1), lgg(c_2, d_2), \ldots) = \bigwedge \{ lgg(r_1, r_2) \}$$

for every pair $r_1, r_2$ of ground instances from each relation in the background knowledge.

RLGG Example

This figure depicts two scenes $s_1$ and $s_2$ and may be described by the predicates Scene/1, On/3, Left-of/2, Circle/1, Square/1 and Triangle/1.

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Ground Instances (tuples)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene</td>
<td>${ &lt; s_1 &gt;, &lt; s_2 &gt; }$</td>
</tr>
<tr>
<td>On</td>
<td>${ &lt; s_1, a, b &gt;, &lt; s_2, f, c &gt; }$</td>
</tr>
<tr>
<td>Left-of</td>
<td>${ &lt; s_1, b, c &gt;, &lt; s_2, d, e &gt; }$</td>
</tr>
<tr>
<td>Circle</td>
<td>${ &lt; a &gt;, &lt; f &gt; }$</td>
</tr>
<tr>
<td>Square</td>
<td>${ &lt; b &gt;, &lt; d &gt; }$</td>
</tr>
<tr>
<td>Triangle</td>
<td>${ &lt; c &gt;, &lt; e &gt; }$</td>
</tr>
</tbody>
</table>
To compute RLGG of the two scenes generate the clause:

\[
\text{Scene}(\text{lgg}(s_1, s_2)) \leftarrow \\
\text{On}(\text{lgg}(s_1, s_2), \text{lgg}(a, f), \text{lgg}(b, e)), \\
\text{Left-of}(\text{lgg}(s_1, s_2), \text{lgg}(b, d), \text{lgg}(c, e)), \\
\text{Circle}(\text{lgg}(a, f)), \\
\text{Square}(\text{lgg}(b, d)), \\
\text{Triangle}(\text{lgg}(c, e))
\]

Compute LGGs to introduce variables into the final clause:

\[
\text{Scene}(A) \leftarrow \\
\text{On}(A, B, C), \\
\text{Left-of}(A, D, E), \\
\text{Circle}(B), \\
\text{Square}(D), \\
\text{Triangle}(E)
\]

**Refinement Operators**

Propositional subsumption — clauses are sets of literals.

E.g., \(\text{flies} \leftarrow \text{bird}\), \(\text{normal}\) can be represented as the set \{\text{flies}, \neg \text{bird}, \neg \text{normal}\}.

In a propositional representation, one clause is more general than the other if it contains a subset of its literals.

For first-order atoms, one atom \(a_1\) is more general than another \(a_2\) if there is a substitution \(\theta\) such that \(a_1\theta \subseteq a_2\).

A **refinement operator** takes one atom (clause) and produces another such that the first atom subsumes the second.

For first-order atoms, **ideal** refinement operators can be found (see tutorial notes).

**From Propositional to First-order Representations**

<table>
<thead>
<tr>
<th>Day</th>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Wind</th>
<th>Play Tennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D2</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D3</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D4</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D5</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D6</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D7</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D8</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D9</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D10</td>
<td>Rain</td>
<td>Mild</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D11</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D12</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D13</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D14</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
</tbody>
</table>
Use a single relation (the \textit{target} relation):

\begin{verbatim}
play_tennis(Day,Outlook,Temperature,Humidity,Wind,PlayTennis).
\end{verbatim}

Training data:

\begin{verbatim}
play_tennis(d1,sunny,hot,high,weak,no).
\end{verbatim}

Hypothesis (complete and correct for examples):

\begin{verbatim}
play_tennis(Day,overcast,Temperature,Humidity,Wind,yes).
play_tennis(Day,rain,Temperature,Humidity,weak,yes).
play_tennis(Day,sunny,Temperature,normal,Wind,yes).
\end{verbatim}

Multiple relations define the \textit{target} w.r.t. \textit{background knowledge}:

\begin{verbatim}
play_tennis(Day,PlayTennis).
\end{verbatim}

Training data:

\begin{verbatim}
play_tennis(d1,no).
\end{verbatim}

Hypothesis (complete and correct for examples):

\begin{verbatim}
play_tennis(Day,yes) :- outlook(Day,overcast).
play_tennis(Day,yes) :- outlook(Day,rain), wind(Day,weak).
play_tennis(Day,yes) :- outlook(Day,sunny), humidity(Day,normal).
\end{verbatim}

Michalski's Trains

\begin{tabular}{ll}
\textbf{Eastbound trains} & \textbf{Westbound trains} \\
\end{tabular}

Michalski's Trains: \textit{background knowledge}

Declare types:

\begin{verbatim}
train(east1). train(east2). train(east3). ...
train(west6). train(west7). train(west8). ...
shape(hexagon). shape(rectangle). shape(triangle).
\end{verbatim}
Define all the trains:

% eastbound train 1
has_car(east1,car_11). long(car_11).
wheels(car_11,2). load(car_11,rectangle,3).

has_car(east1,car_12). short(car_12).
wheels(car_12,2). shape(car_12,rectangle).
...

% eastbound train 2
has_car(east2,car_21). short(car_21).
open_car(car_21). load(car_21,triangle,1).
...

Logically, the negative examples are instances (here the trains west1, west2, etc.) for which the target predicate (here eastbound/1) is false.

Learned using Aleph in SWI Prolog:

[clauses constructed] [70]
[search time] [0.01]
[best clause]
eastbound(A) :-
    has_car(A, B), short(B), closed(B).
[pos cover = 5 neg cover = 0] [pos-neg] [5]
true.

?-

Learning First Order Rules

- to learn logic programs we can adopt propositional rule learning methods
- the target relation is clause head, e.g. ancestor/2
  - think of this as the consequent
- the clause body is constructed using predicates from background knowledge
  - think of this as the antecedent
- unlike propositional rules first order rules can have
  - variables
  - tests on more than one variable at a time
  - recursion
- learning is set up as a search through the hypothesis space of first order rules
Example: 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
Can learn graph-type representations.

FOIL(Target predicate, Predicates, Examples)
Pos := positive Examples
Neg := negative Examples
while Pos, do
   // Learn a NewRule
   NewRule := most general rule possible
   NewRuleNeg := Neg
   while NewRuleNeg, do
      // Add a new literal to specialize NewRule
      Candidate_literals := generate candidates
      Best_literals := argmax_L∈Candidate_literals |Foil_Gain(L, NewRule)|
      add Best_literals to NewRule preconditions
      NewRuleNeg := subset of NewRuleNeg that satisfies NewRule preconditions
   Learned_rules := Learned_rules + NewRule
   Pos := Pos − {members of Pos covered by NewRule}
Return Learned_rules

Specializing Rules in FOIL

Learning rule: \( P(x_1, x_2, \ldots, x_k) \leftarrow L_1 \ldots L_n \)
Candidate specializations add new literal of form:

- \( Q(v_1, \ldots, v_r) \), where at least one of the \( v_i \) in the created literal must already exist as a variable in the rule.
- \( Equal(x_j, x_k) \), where \( x_j \) and \( x_k \) are variables already present in the rule
- The negation of either of the above forms of literals

Completeness and Consistency (Correctness)

<table>
<thead>
<tr>
<th>( \mathcal{H} ): complete, consistent</th>
<th>( \mathcal{H} ): incomplete, consistent</th>
</tr>
</thead>
<tbody>
<tr>
<td>( covers(\mathcal{H}, \mathcal{E}) )</td>
<td>( covers(\mathcal{H}, \mathcal{E}) )</td>
</tr>
<tr>
<td>( \mathcal{E}^+ )</td>
<td>( \mathcal{E}^- )</td>
</tr>
<tr>
<td>( \mathcal{E}^+ )</td>
<td>( \mathcal{E}^- )</td>
</tr>
<tr>
<td>( - )</td>
<td>( - )</td>
</tr>
</tbody>
</table>
Completeness and Consistency (Correctness)

- $\mathcal{H}$: complete, inconsistent

- $\mathcal{H}$: incomplete, inconsistent

Information Gain in FOIL

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

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 $R + L$

Variable Bindings

- A substitution replaces variables by terms
- Substitution $\theta$ applied to literal $L$ is written $L\theta$
- If $\theta = \{x/3, y/z\}$ and $L = P(x, y)$ then $L\theta = P(3, z)$

FOIL bindings are substitutions mapping each variable to a constant:

$$GrandDaughter(x, y) \leftarrow$$

With 4 constants in our examples we have 16 possible bindings:

$$\{x/Victor, y/Sharon\}, \{x/Victor, y/Bob\}, \ldots$$

With 1 positive example of GrandDaughter, other 15 bindings are negative:

$$GrandDaughter(Victor, Sharon)$$

Information Gain in FOIL

Note

- $- \log_2 \frac{p_0}{p_0 + n_0}$ is minimum number of bits to identify an arbitrary positive binding among the bindings of $R$
- $- \log_2 \frac{n_0}{p_0 + n_0}$ is minimum number of bits to identify an arbitrary positive binding among the bindings of $R + L$
- $Foil\textunderscore Gain(L, R)$ measures the reduction due to $L$ in the total number of bits needed to encode the classification of all positive bindings of $R$
Learning with FOIL

Target Predicate: ancestor

FOIL as a propositional learner

- target predicate is usual form of class value and attribute values
  - \( Class_1(V_1, V_2, \ldots, V_m), Class_2(V_1, V_2, \ldots, V_m) \ldots \)
- literals restricted to those in typical propositional learners
  - \( V_i = const, V_i > num, V_i \leq num \)
- plus extended set
  - \( V_i = V_j, V_i \geq V_j \)
- FOIL results vs C4.5
  - accuracy competitive, especially with extended literal set
  - FOIL required longer computation
  - C4.5 more compact, i.e. better pruning

FOIL learns Prolog programs from examples

- from I. Bratko’s book “Prolog Programming for Artificial Intelligence”
- introductory list programming problems
- training sets by randomly sampling from universe of 3 and 4 element lists
- FOIL learned most predicates completely and correctly
  - some predicates learned in restricted
  - some learned in more complex form than in book
  - most learned in few seconds, some much longer

Completeness and Correctness

New clause: ancestor(X,Y) :-.
Best antecedent: parent(X,Y)   Gain: 31.02
Learned clause: ancestor(X,Y) :- parent(X,Y).

New clause: ancestor(X,Y) :-.
Best antecedent: parent(Z,Y)     Gain: 13.65
Best antecedent: ancestor(X,Z)  Gain: 27.86
Learned clause: ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).

Definition: ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).
 FOIL learns Prolog programs from examples

member(E,L)  E is an element of list L
conc(L1,L2,L3) appending L1 to L2 gives list L3
member(E,L)  as for member with conc available
first(L1)    E is the first element of L1
list(L1)     list but without using conc
diff(L1,L2,L3) deleting an occurrence of E from L1 gives L2
member2(E,L) as for member with diff available
insert(L1,E,L2) inserting E somewhere in L1 gives L2
subset(L1,L2) L1 is a sublist of L2
permute(L1,L2) L2 is a permutation of list L1
even/odd(length) L has an even/odd number of elements (both relations to be defined)
reverse(L1)  L2 is the reverse of list L1
palindrome(L) list L is a palindrome
palindrome2(L) as above, but not using reverse
shift(L1,L2) rotating elements of L1 to the left gives L2
translate(L1,L2) L2 is the result of translating L1 using an
element-to-element mapping
subset(S1,S2) S2 is a subset of set S1
distinct(L1,L2,L3) L3 contains the odd-numbered elements of L1
L3 contains the even-numbered elements of L1

Determinate Literals

- adding a new literal \( Q(X, Y) \) where \( Y \) is the unique value for \( X \)
- this will result in zero gain!
- FOIL gives a small positive gain to literals introducing a new variable
- BUT there may be many such literals

Identifying document components

- Problem: learn rules to locate logical components of documents
- documents have varying numbers of components
- relationships (e.g. alignment) between pairs of components
- inherently relational task
- target relations to identify sender, receiver, date, reference, logo.
Text applications of first-order logic in learning

Q: when to use first-order logic in machine learning?
A: when relations are important.

Representation for text

Example: text categorization, i.e. assign a document to one of a finite set of categories.

Propositional learners:

- use a “bag-of-words”, often with frequency-based measures
- disregards word order, e.g. equivalence of
  
  That’s true, I did not do it
  That’s not true, I did do it

First-order learners: word-order predicates in background knowledge

\[
\text{has_word}(\text{Doc}, \text{Word}, \text{Pos})
\]

\[
\text{Pos1 < Pos2}
\]
Learning information extraction rules

What is information extraction? Fill a pre-defined template from a given text.

Partial approach to finding meaning of documents.

**Given:** examples of texts and filled templates

**Learn:** rules for filling template slots based on text

*Sample Job Posting*

**Subject:** US-TN-SOFTWARE PROGRAMMER  
**Date:** 17 Nov 1996 17:37:29 GMT  
**Organization:** Reference.Com Posting Service  
**Message-ID:** 56nigp$mrs@bilbo.reference.com

SOFTWARE PROGRAMMER

Position available for Software Programmer experienced in generating software for PC-Based Voice Mail systems. Experienced in C Programming. Must be familiar with communicating with and controlling voice cards; preferable Dialogic, however, experience with others such as Rhetoric and Natural Microsystems is okay. Prefer 5 years or more experience with PC Based Voice Mail, but will consider as little as 2 years. Need to find a Senior level person who can come on board and pick up code with very little training. Present Operating System is DOS. May go to OS-2 or UNIX in future.

Please reply to:  
Kim Anderson  
AdNET  
(901) 458-2888 fax  
kimander@memphisonline.com

*Example filled template*

<table>
<thead>
<tr>
<th>id</th>
<th>56nigp$<a href="mailto:mrs@bilbo.reference.com">mrs@bilbo.reference.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>SOFTWARE PROGRAMMER</td>
</tr>
<tr>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>company</td>
<td></td>
</tr>
<tr>
<td>recruiter</td>
<td></td>
</tr>
<tr>
<td>state</td>
<td>TN</td>
</tr>
<tr>
<td>city</td>
<td></td>
</tr>
<tr>
<td>country</td>
<td>US</td>
</tr>
<tr>
<td>language</td>
<td>C</td>
</tr>
<tr>
<td>platform</td>
<td>PC</td>
</tr>
<tr>
<td>application area</td>
<td>Voice Mail</td>
</tr>
<tr>
<td>req years experience</td>
<td>2</td>
</tr>
<tr>
<td>desired years experience</td>
<td>5</td>
</tr>
<tr>
<td>req degree</td>
<td></td>
</tr>
<tr>
<td>desired degree</td>
<td></td>
</tr>
<tr>
<td>post date</td>
<td>17 Nov 1996</td>
</tr>
</tbody>
</table>
A learning method for Information Extraction

Rapier (Califf and Mooney, 2002) is an ILP-based approach which learns information extraction rules based on regular expression-type patterns.

Pre-Filler Patterns: what must match before filler
Filler Patterns: what the filler pattern is
Post-Filler Patterns: what must match after filler

Algorithm uses a combined bottom-up (specific-to-general) and top-down (general-to-specific) approach to generalise rules.

syntactic analysis: Brill part-of-speech tagger
semantic analysis: WordNet (Miller, 1993)

Example rules from text to fill the city slot in a job template:

"... located in Atlanta, Georgia."
"... offices in Kansas City, Missouri."

Pre-Filler Pattern | Filler Pattern | Post-Filler Pattern
--- | --- | ---
1) word: in | 1) list: max length: 2 | 1) word: ,
tag: in | tag: npn | tag: ;
semantic: state | semantic: state

where npn denotes a proper noun (syntax) and state is a general label from the WordNet ontology (semantics).

Progol

**Progol**: Reduce combinatorial explosion by generating most specific acceptable \( h \) as lower bound on search space

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 \( h \) s.t. \( B \land h \land x_i \vdash f(x_i) \)
     - actually, considers only \( k \)-step entailment
3. Conduct general-to-specific search bounded by specific hypothesis \( h_i \), choosing hypothesis with minimum description length

Protein structure

**Positive(12)**

**Negative(12)**
fold('Four-helical up-and-down bundle', P) :-
    helix(P, H1),
    length(H1, hi),
    position(P, H1, Pos),
    interval(1 <= Pos <= 3),
    adjacent(P, H1, H2),
    helix(P, H2).

“The protein P has fold class 'Four-helical up-and-down bundle' if it contains a long helix H1 at a secondary structure position between 1 and 3 and H1 is followed by a second helix H2”.

Immunoglobulin:-
Has antiparallel sheets B and C; B has 3 strands, topology123; C has 4 strands, topology 2134.

TIM barrel:-
Has between 5 and 9 helices; Has a parallel sheet of 8 strands.

SH3:-
Has an antiparallel sheet B, C and D are the 1st and 4th strands in the sheet B respectively. C and D are the end strands of B and are 4.360 (+/- 2.18) angstroms apart. D contains a proline in the c-terminal end.
Inductive Programming

FlashFill: An Excel 2013 feature that automates repetitive string transformations using examples. Once the user performs one instance of the desired transformation (row 2, col. B) and proceeds to transforming another instance (row 3, col. B), FlashFill learns a program:

\[
\text{Concatenate(ToLower(Substring(v,WordToken,1)),
ToLower(Substring(v,WordToken,2)))}
\]

that extracts the first two words in input string \(v\) (col. A), converts them to lower case, and concatenates them separated by a space character.


Summary

- can be viewed as an extended approach to rule learning
- BUT: much more ...
- learning in a general-purpose programming language
- use of rich background knowledge
- incorporate arbitrary program elements into clauses (rules)
- background knowledge can grow as a result of learning
- control search with declarative bias
- learning probabilistic logic programs