Inductive Logic Programming

Ganesh Ramakrishnan

Adjunct Professor, CSE Dept
&
IBM India Research Labs, Delhi
What is ILP

- Inductive Logic Programming?
- Inductive Logic Programming?

OR

Machine Learning

Applications
Statistical techniques

ILP

Logic Programming

Representation
Theory
Implementation

Theory, Implementation and Application of programs that construct logic programs from examples
Machine Learning
Programs that hypothesize general descriptions from sample data

Sample Data

Hypothesis

<table>
<thead>
<tr>
<th>X1</th>
<th>X3</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

X8 > 0.3

No

Yes

X20 = n

Y = +

Instances of some sorted/unsorted lists of numbers

...
Logic Programming

- Study of using symbolic logic as a programming language
- Specification = Programming

Logic program:
\[
\forall X, Y \; \text{grandfather}(X, Y) \leftarrow \exists Z \; (\text{father}(X, Z), \text{parent}(Z, Y))
\]
father(henry,jane) ←
father(henry,joe) ←
parent(jane,john) ←
parent(joe,robert) ←

Derived facts:
grandfather(henry,john) ←
grandfather(henry,robert) ←
Logical Reasoning: 3 types

- Given preconditions $\alpha$, post-conditions $\beta$ and the rule $R1: \alpha \therefore \beta$ ($\alpha$ therefore $\beta$).
  - **Deduction** means determining $\beta$. It is using the rule and its preconditions to make a conclusion.
  - **Induction** means determining $R1$. It is learning $R1$ after numerous examples of $\beta$ and $\alpha$.
  - **Abduction** means determining $\alpha$. It is using the post-condition and the rule to assume that the precondition could explain the postcondition ($\beta \land R1 \Rightarrow \alpha$).
First Order Logic: Primer
First-Order Logic: Primer

- **Constant**: Objects in the domain  
  E.g.: Anna,

- **Variable**: Ranges over objects  
  E.g.: x

- **Functions**: Takes a tuple of objects and returns an object  
  E.g.: MotherOf(x), Friends(x, y)

- **Predicates**: Represents either the property of an object or relationship between objects  
  E.g.: IsTall(x), Friends(x, y)
First-Order Logic

- **Terms**: A constant, variable or functional expression (a function applied to a tuple of terms)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Term?</th>
</tr>
</thead>
<tbody>
<tr>
<td>peter</td>
<td>✓</td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>log(X)</td>
<td>✓</td>
</tr>
<tr>
<td>son(peter, peter)</td>
<td>x</td>
</tr>
<tr>
<td>log(son(peter, peter))</td>
<td>x</td>
</tr>
<tr>
<td>sin(log(cos(X/2)))</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **Atoms**: Predicate symbol applied to a tuple of terms
  - son(spock, sarek)

- **Clause**: Statements of the form \( p_1 \lor p_2 \ldots \leftarrow q_1 \land q_2 \)
  - **Definite clause**: Head has 1 atom without a negation and body has no negation.

- **Datalog** = First Order Logic – Function symbols
  - Term Datalog was coined in the mid 1980's by a group of researchers interested in database theory.
First-Order Logic

- **World/Domain** (Hebrand Interpretation): Assignment of truth values to all ground predicates
  - If we take all predicates and replace variables with constants, we will get a large number of Boolean variables.
  - Construction of world is concerned with truth assignments to these Boolean variables.
  - Later, we will be interested in probability distributions over these assignments.

- **Propositionalization:**
  Create all ground atoms and clauses. The resultant set is called the *Herbrand Universe*
Model Theory

- **(Herbrand) Model**
  - An interpretation that gives the value \textit{TRUE} for a formula is called a \textit{model} for that formula

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( p \leftrightarrow q )</th>
<th>Model for ( p \leftrightarrow q )?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>( f )</td>
<td>( t )</td>
<td>( f )</td>
<td>( \times )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
<td>( t )</td>
<td>( \checkmark )</td>
</tr>
<tr>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
<td>( \checkmark )</td>
</tr>
</tbody>
</table>

- **Valid Formulae**
  - Formulae for which \textit{every} interpretation is a model are said to be \textit{valid}

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( (p \leftrightarrow q) \land q )</th>
<th>( p \leftrightarrow (p \leftrightarrow q) \land q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( f )</td>
<td>( t )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
</tbody>
</table>

- **Model Theory**
  - Concerned with attributing meaning to logical sentences
Model Theory (contd)

- Satisfiability
  - A formula is said to be *satisfiable* if it has at least 1 model. Otherwise it is said to be *unsatisfiable*.
- The set of all *Herbrand* models for a definite-clause program $P$ is partially ordered by $\subseteq$ and forms a lattice.
Reasoning: Primer
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/formula from the right and make it a condition
Proof Procedures In Logic

- **Concern**
  - Searching spaces efficiently, keeping in mind soundness and completeness

- **Soundness**
  - Anything that is derived *should* be a logical consequence

- **Completeness**
  - *Any* logical consequence should be derivable
Computation and Search Rules

- Typical logic problem: Solve queries of the form $l_1, l_2, \ldots , l_n$ where the $l_i$ are literals

- Two issues:
  - Which literal of the $l_i$ should be solved first?
    The rule governing this is called the *computation* rule
    (determines a tree of choices)
  - 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
    (determines the order in which this tree is searched)
Resolution and Unification

- Given: Program $P$ and a query: $l_1, l_2, \ldots, l_{j-1}, l_j, \ldots, l_n$?
  1. Use the computation rule to select $l_j$.
  2. Use the search rule to select a clause: $l_j \leftarrow b_1, b_2, \ldots, b_k$ in $P$ that can solve $l_j$. If none found, STOP.
  3. Solve the query: $l_1, l_2, \ldots, l_{j-1}, b_1, b_2, \ldots, b_k, \ldots l_n$?
    - The step of replacing the literal selected with the literals comprising the body of the clause is an application of the rule of inference known as **resolution**

- The head of the clause selected does not have to match **exactly** the literal selected. It will be enough if the two can **unify**

- **Unification**: There is some substitution of variables for terms in the two literals that makes them the same
  - Unification is “join” with respect to a **specialisation order**
  - E.g: $f(g(A), A) = f(B, xyz)$: Unifies $A$ with the atom $xyz$ and $B$ with the term $g(xyz)$
Example for Datalog

1. \( \text{gparent}(X, Z) \leftarrow \text{parent}(X, Y), \text{parent}(Y, Z) \)
2. \( \text{parent}(\text{tom}, \text{jo}) \leftarrow \)
3. \( \text{parent}(\text{jo}, \text{bob}) \leftarrow \)
4. \( \text{parent}(\text{jo}, \text{jim}) \leftarrow \)

**Rightmost literal first computation rule:**

```
gparent(\text{tom}, \text{Z})\
  \text{1 \{X/tom\}}
  \text{parent(\text{tom}, \text{Y}), parent(\text{Y}, \text{Z})?}
  \text{2 \{Y/tom, Z/jo\}}
  \text{parent(\text{tom}, \text{tom})?}
    \text{FAIL}
    \text{SUCCESS}

\text{parent(\text{tom}, \text{jo})? parent(\text{tom}, \text{jo})?}
  \text{3 \{Y/jo, Z/bob\}}
  \text{4 \{Y/jo, Z/jim\}}
```

"substitutions"
Example for Datalog

1. \textit{less}\_\textit{than}(X, Y) \leftarrow \textit{succ}(Y, X)
2. \textit{less}\_\textit{than}(X, Y) \leftarrow \textit{succ}(Z, Y), \textit{less}\_\textit{than}(Z, Y)
3. \textit{succ}(\textit{two}, \textit{one}) \leftarrow
4. \textit{succ}(\textit{three}, \textit{two}) \leftarrow

Leftmost literal rule for the query

\textit{less}\_\textit{than}(\textit{one}, Y)

```
less_than(one, Y)
  
  less_than(one, Y)?
    1 [X/one]
  
  succ(Y, one)?
    3 [Y/two]

SUCCESS

less_than(two, Y)
  
  less_than(two, Y)?
    1 [X'/two, Y'/Y]
    4 [Y/three]

SUCCESS

less_than(three, Y)?
  
  less_than(three, Y)?
    1 [X''/three, Y''/Y]
  
  succ(Y, three)?
    FAIL

FAIL
```
One way to search the trees obtained so far is depth-first, left-to-right
- Since clauses that appear first (reading top to bottom) in the program have been drawn on the left, this search rule selects clauses in order of appearance in the program

Most logic programs are executed using the following:
- Computation rule. Leftmost literal first
- Search rule. Depth first search for clauses in order of appearance
Inference in First-Order Logic

- Traditionally done by theorem proving
  - E.g.: Prolog uses resolution
- More recently…..
  - Propositionalization followed by model checking turns out to be much faster
  - Comes as a surprise, because expected to be inefficient, since not “lifted”
  - However, in many domains, it is very fast.
- **Model checking**: Satisfiability testing
  - Two main families of satisfiability solvers:
  - **Backtracking** (Typical example is DPLL)
  - **Stochastic local search** (Typical example is WalkSAT)
Satisfiability

- **Input:**
  - Set of clauses
  - Convert KB to conjunctive normal form (CNF) after propositionalization
  - Every propositional formula can be converted into an equivalent formula that is in CNF using rules about logical equivalences:
    - The Double Negative Law
    - The De Morgan's laws
    - The distributive Law.

- **Output:**
  - Truth assignment that satisfies all clauses, OR
  - Failure (if no truth assignment exists)

- The paradigmatic NP-complete problem

- **Solution:** Search
Satisfiability

- **Parameters:**
  - #Clauses: More clauses give more constraints
  - #Variables: More variables give more freedom

- **Key point:**
  - Most SAT problems are under-constrained and actually easy
    - In many cases, any random solution satisfies all clauses
  - Though exponentially hard in worst case

- **Hard region:** Over-constrained
  - Small region of parameter space
  - Narrow range of #Clauses / #Variables
  - For random 3-sat problems, the hard region is approx for #Clauses / #Variables > 4 (*area of research*)
Backtracking

- **Basic Idea:**
  - Assign truth values by depth-first search
    1. Start off with no truth values assigned
- Assigning a variable deletes false literals and satisfied clauses
- Empty set of clauses: Success
- Empty clause: Failure
- Additional improvements:
  - **Unit propagation** (unit clause forces truth value)
  - **Pure literals** (same truth value everywhere)
DPLL example

C1: \((a \lor b)\)
C2: \((\neg a \lor \neg b)\)
C3: \((a \lor \neg c)\)
C4: \((c \lor d \lor e)\)
C5: \((d \lor \neg e)\)
C6: \((\neg d \lor \neg f)\)
C7: \((f \lor e)\)
C8: \((\neg f \lor \neg e)\)

Legend

\[\rightarrow\] false

\[\rightarrow\] true

\(a=\text{false by branching}\)

\(a=\text{false by pure symbol}\)

\(a=\text{true by an unit clause}\)
C1: (a ∨ b)
C2: (¬a ∨ ¬b)
C3: (a ∨ ¬c)
C4: (c ∨ d ∨ e)
C5: (d ∨ ¬e)
C6: (¬d ∨ ¬f)
C7: (f ∨ e)
C8: (¬f ∨ ¬e)

DPLL example

Pure Symbol?
Yes, b in C1 is pure
No pure symbol

Unit Clause?
No unit clause
C4 is a unit clause
C5 is unsatisfied, Early termination
Backtrack up to the last branching: d = false
DPLL example

C1: (a ∨ b)
C2: (¬a ∨ ¬b)
C3: (a ∨ ¬c)
C4: (c ∨ d ∨ e)
C5: (d ∨ ¬e)
C6: (¬d ∨ ¬f)
C7: (f ∨ e)
C8: (¬f ∨ ¬e)

C6 is an unit clause
f is pure

Formula Satisfied!

branching
pure symbol
unit clause
false
true
Exercise

- Find a satisfying assignment using DPLL

\((\neg a \lor b) \land (\neg a \lor \neg b \lor c)\)

\((\neg c \lor d \lor \neg e) \land (a \lor c)\)

\((\neg d \lor \neg f) \land (a \lor c)\)

\((e \lor \neg f)\)
The DPLL Algorithm

if $CNF$ is empty then
    return $true$
else if $CNF$ contains an empty clause then
    return $false$
else if $CNF$ contains a pure literal $x$ then
    return $DPLL(CNF(x))$
else if $CNF$ contains a unit clause $\{u\}$ then
    return $DPLL(CNF(u))$
else
    choose a variable $x$ that appears in $CNF$
    if $DPLL(CNF(x)) = true$ then return $true$
    else return $DPLL(CNF(\neg x))$
Stochastic Local Search

- Uses complete assignments instead of partial
- Start with random state
- Flip variables in unsatisfied clauses
- Hill-climbing: Minimize # unsatisfied clauses
- Avoid local minima: Random flips
- Multiple restarts
The WalkSAT Algorithm

\[
\text{for } i \leftarrow 1 \text{ to max-tries do }
\]
\[
\quad \text{solution} = \text{random truth assignment }
\]
\[
\text{for } j \leftarrow 1 \text{ to max-flips do }
\]
\[
\quad \text{if all clauses satisfied then}
\]
\[
\quad \quad \text{return solution}
\]
\[
\quad c \leftarrow \text{random unsatisfied clause}
\]
\[
\quad \text{with probability } p
\]
\[
\quad \quad \text{flip a random variable in } c
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{flip variable in } c \text{ that maximizes number of satisfied clauses}
\]
\[
\text{return failure}
\]
Rule Induction
Rule Induction

- **Given**: Set of positive and negative examples of some concept
  - **Example**: \((x_1, x_2, \ldots, x_n, y)\)
  - \(y\): concept (Boolean)
  - \(x_1, x_2, \ldots, x_n\): attributes (assume Boolean)

- **Goal**: Induce a set of rules that cover all positive examples and no negative ones
  - **Rule**: \(x_a \land x_b \land \ldots \Rightarrow y\) (\(x_a\): Literal, i.e., \(x_i\) or its negation)
  - Same as **Horn clause**: Body \(\Rightarrow\) Head
  - Rule \(r\) **covers** example \(x\) iff \(x\) satisfies body of \(r\)

- **Eval\((r)\)**: Accuracy, info. gain, coverage, support, etc.
Learning a Single Rule

\[ \text{head} \leftarrow y \]
\[ \text{body} \leftarrow \emptyset \]
\[ \text{repeat} \]
\[ \text{for each literal } x \]
\[ r_x \leftarrow r \text{ with } x \text{ added to } \text{body} \]
\[ \text{Eval}(r_x) \]
\[ \text{body} \leftarrow \text{body} \wedge \text{best } x \]
\[ \text{until no } x \text{ improves } \text{Eval}(r) \]
\[ \text{return } r \]
Learning a Set of Rules

\[
\begin{align*}
R & \leftarrow \emptyset \\
S & \leftarrow \text{examples} \\
\text{repeat} \\
& \quad \text{learn a single rule } r \\
& \quad R \leftarrow R \cup \{ r \} \\
& \quad S \leftarrow S - \text{positive examples covered by } r \\
\text{until } S & = \emptyset \\
\text{return } R
\end{align*}
\]
First-Order Rule Induction

- \( y \) and \( x_i \) are now predicates with arguments
  E.g.: \( y \) is \( \text{Ancestor}(x,y) \), \( x_i \) is \( \text{Parent}(x,y) \)
- Literals to add are predicates or their negations
- Literal to add must include at least one variable already appearing in rule
- Adding a literal changes \# groundings of rule
  E.g.: \( \text{Ancestor}(x,z) \land \text{Parent}(z,y) \Rightarrow \text{Ancestor}(x,y) \)
- \( \text{Eval}(r) \) must take this into account
  E.g.: Multiply by \# positive groundings of rule
  still covered after adding literal
“Inductive” Logic Programming

(Sample data)

**Examples:**
grandfather(henry,john) ←
grandfather(henry,robert) ←

+  

**Background:**
father(henry,jane) ←
father(henry,joe) ←
parent(jane, john) ←
parent(joe, robert) ←

**Hypothesis:**
\[ \forall X, Y \text{ grandfather}(X, Y) \leftarrow \exists Z \text{ (father}(X, Z) \text{, parent}(Z, Y)) \]

(A logic program)
More “Interesting” ILP

**Examples:**
- Some carcinogenic chemicals
- Some non-carcinogenic chemicals

**Background:**
- Molecular structure of chemicals
- General chemical knowledge

**Hypothesis:**
\[ \forall X \text{ carcinogenic}(X) \leftarrow \ldots \]
\[ \ldots \]
\[ \ldots \]

- 1000’s
- 10,000’s
- 10’s
Induction by inverting deduction

- First investigated in depth mathematically by the 19th century political economist and philosopher of science Stanley Jevons
- From Jevons' book on inductive inference:
  - *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; the integral calculus, for instance, is infinitely more difficult than the differential calculus of which it is the inverse. Similarly, it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction;* ...
Hypothesis formation and justification

- **Abduction.** Process of hypothesis formation.

- **Justification.** The degree of belief assigned to a hypothesis given a certain amount of evidence.
Specific logical setting for abduction

\[ B = C_1 \land C_2 \land \ldots \quad \text{Background} \]
\[ E = E^+ \land E^- \quad \text{Examples} \]
\[ E^+ = e_1 \land e_2 \land \ldots \quad \text{Positive Examples} \]
\[ E^- = f_1 \land f_2 \land \ldots \quad \text{Negative Examples} \]
\[ H = D_1 \land D_2 \land \ldots \quad \text{Hypothesis} \]

**Prior Satisfiability.** \( B \land E^- \not\models \square \)

**Posterior Satisfiability.** \( B \land H \land E^- \not\models \square \)

**Prior Necessity.** \( B \not\models E^+ \)

**Posterior Sufficiency.** \( B \land H \models E^+ \),
\[ B \land D_i \models e_1 \lor e_2 \lor \ldots \]
Acknowledgement

Some Slides borrowed from

- Ashwin Srinivasan
- Pedro Domingos tutorial on Statistical Relational Learning at ICML’07