Learning Rules

In learning rules, we are interested in learning rules of the form:

$$\textbf{if } A_1 \land A_2 \land \ldots \textbf{ then } C$$

where $A_1, A_2, \ldots$ are the preconditions/constraints/body/antecedents of the rule and $C$ is the postcondition/head/consequent of the rule.

A first-order rule can contain variables, as in:
$$\textbf{if } \text{Parent}(x, z) \land \text{Ancestor}(z, y) \textbf{ then } \text{Ancestor}(x, y)$$

A Horn clause is a rule with no negations.

Learning Rules from Decision Trees

Each path in a decision tree from root to a leaf:

- if \text{windy} \ then \ \text{true}
- if \text{outlook} \ then \ \text{false}
- if \text{humidity} \ then \ \text{sunny}
- if \text{good} \ then \ \text{overcast}
- if \text{bad} \ then \ \text{rain}
- if \text{temp} \ then \ \text{hot}
- if \text{mild} \ then \ \text{cool}
- if \text{cool} \ then \ \text{high}
- if \text{good} \ then \ \text{normal}
- if \text{bad} \ then \ \text{bad}
- if \text{good} \ then \ \text{good}
- if \text{rain} \ then \ \text{overcast}
- if \text{sunny} \ then \ \text{rain}
- if \text{???} \ then \ \text{false}
- if \text{bad} \ then \ \text{true}
- if \text{???} \ then \ \text{true}
- if \text{good} \ then \ \text{false}

can be converted to a rule, e.g.:

$$\textbf{if } \neg \text{windy} \land \text{hot} \land \text{sunny} \textbf{ then } \text{bad}$$

Why? Rules are considered more readable/understandable. Different pruning decisions can be made for different rules.
Rule Post-Pruning

Rule post-pruning removes conditions to improve error.

\begin{align*}
\text{if } \text{windy} \land \text{sunny} \land \text{high} & \text{ then bad} \\
\text{if } \text{windy} \land \text{sunny} \land \text{normal} & \text{ then good} \\
\text{if } \text{windy} \land \text{overcast} & \text{ then good} \\
\text{if } \text{windy} \land \text{rain} & \text{ then bad} \\
\text{if } \neg \text{windy} \land \text{hot} \land \text{sunny} & \text{ then bad} \\
\text{if } \neg \text{windy} \land \text{hot} \land \text{overcast} & \text{ then good} \\
\text{if } \neg \text{windy} \land \text{mild} \land \text{sunny} & \text{ then bad} \\
\text{if } \neg \text{windy} \land \text{mild} \land \text{rain} & \text{ then good} \\
\text{if } \neg \text{windy} \land \text{cool} & \text{ then good} \\
\end{align*}

Rule Ordering

Order rules by accuracy and coverage.

4 if overcast then good
3 if \neg \text{windy} \land \text{rain} then good
3 if sunny \land \text{high} then bad
2 if sunny \land \text{normal} then good
2 if \neg \text{windy} \land \text{cool} then good
2 if wind \land \text{rain} then bad
2 if hot \land \text{sunny} then bad
1 if \neg \text{windy} \land \text{mild} \land \text{sunny} then bad
Sequential Covering Algorithms

Basic Algorithmic Idea:

1. Learn one good rule.
2. Remove the examples covered by the rule.
3. Repeat until no examples are left.

The book focuses on rules to cover the positive examples. This is because negative/false is the default answer in logic programming languages like Prolog.

However, the same algorithms can be applied to negative examples, or to multi-class datasets.

Considerations

A good rule has low error (entropy) and high coverage.

A conjunction of two or more attribute tests might be required to obtain a good rule.

A depth-first greedy search (start from no tests, then repeatedly add best looking test) can lead down a bad path.

In an example-driven search, choose one example and only add tests that cover that example.

A beem search maintains a list of the $k$ best candidates, i.e., searches $k$ paths at a time instead of just one.
Learn-One-Rule Beam-Search-Algorithm

**Learn-One-Rule**

\[
\begin{align*}
\text{best} & \leftarrow \emptyset, \text{frontier} \leftarrow \{\text{best}\} \\
\text{while} \text{ frontier is not empty} & \\
\text{candidates} & \leftarrow \text{all hypotheses generated by adding a test to a hypothesis in frontier} \\
\text{best} & \leftarrow \text{best}_1 \text{ hypothesis in } \{\text{best}\} \cup \text{candidates} \\
\text{frontier} & \leftarrow \text{best}_2 \ k \text{ hypotheses in candidates} \\
\text{return} & \text{ best}
\end{align*}
\]

“best$_1$” might be error rate.

“best$_2$” might be entropy (different only for multi-class)

Both might require covering a minimum number of exs.

-------------------------------

Learning First-Order Rules

Suppose we are interested in learning concepts involving relationships between objects.

- When one person is an ancestor of another number.
- When one number is less than another number.
- When one node can reach another node in a graph.
- When an element is in a set.

The concept involves intermediate objects and relations.

Examples can’t be written as a fixed number of attributes.
Terminology: Examples of Atoms

To say that an object has a property, e.g., $-3$ is negative, we write \textit{Negative}($-3$), where \textit{Negative} is a predicate and $-3$ is a constant.

To say that multiple objects have a relationship, e.g., Liz is the mother of Charley, we write \textit{Mother}(Liz, Charley), \textit{Mother} is a predicate and \textit{Liz} and \textit{Charley} are constants.

To say that attributes of objects have a relationship, e.g., Liz’s computer is faster than Charley’s, we write \textit{Faster}(\textit{computer}($Liz$), \textit{computer}($Charley$)), where \textit{Faster} is a predicate, \textit{computer} is a function, and \textit{Liz} and \textit{Charley} are constants.

---

Terminology: Examples of Rules

To say that negative numbers are less than positive numbers, we write: \textit{Less}(x, y) \leftarrow \textit{Negative}(x) \land \textit{Positive}(y), where $x$ and $y$ are variables. Do not write it in this way: \textit{Less}(\textit{Negative}(x), \textit{Positive}(y))

The rule for grandmother can be written as:
\textit{Grandmother}(x, y) \leftarrow \textit{Mother}(x, z) \land \textit{Parent}(z, y)
Note that an additional variable $z$ is needed to define the relationship between $x$ and $y$.

Rules can be recursive:

\((x+y = z) \leftarrow (u+v = z) \land (x+1 = u) \land (v+1 = y)\)
where $x+y = z$ is short for \textit{Equal}(\textit{plus}(x, y), z).
Terminology

A term is any constant or variable, or any function applied to terms.

An atom is any predicate applied to terms.

A literal is an atom (positive) or the negation of an atom (negative).

A clause is a disjunction (or) of literals.

A Horn clause has at most one positive literal.

A Horn clause $H \lor \neg L_1 \lor \neg L_2 \lor \ldots$ can be written as $H \leftarrow (L_1 \land L_2 \land \ldots)$

Example

Suppose we want to learn reachability in graphs.

In this graph, the predicate $Edge$ is true for:

$(A, B) \ (B, C) \ (C, D) \ (D, B) \ (D, E)$

and false for:

$(A, A) \ (A, C) \ (A, D) \ (A, E) \ (B, A) \ (B, B) \ (B, D) \ (B, E) \ (C, A) \ (C, B) \ (C, C) \ (C, E) \ (D, A) \ (D, C) \ (D, D) \ (E, A) \ (E, B) \ (E, C) \ (E, D) \ (E, E)$
The predicate $\textit{Reach}$ is true for:

\[
(A, B) \quad (A, C) \quad (A, D) \quad (A, E) \quad (B, B) \quad (B, C) \\
(B, D) \quad (B, E) \quad (C, B) \quad (C, C) \quad (C, D) \quad (C, E) \\
(D, B) \quad (D, C) \quad (D, D) \quad (D, E)
\]

and false for:

\[
(A, A) \quad (B, A) \quad (C, A) \quad (D, A) \quad (E, A) \quad (E, B) \\
(E, C) \quad (E, D) \quad (E, E)
\]

We want literals that make $\textit{Reach}(x, y)$ “more true”. When choosing a literal, the FOIL program maximizes:

\[
p_1 \left( \log_2 \frac{p_1}{p_1 + n_1} - \log_2 \frac{p_0}{p_0 + n_0} \right)
\]

where $p_1$ ($n_1$) is the number of pos. (neg.) examples still covered after adding $L$. $p_0$ and $n_0$ are the before numbers.

---

First Rule for Example

$\textit{Edge}(x, y)$ is true for 5 pos. exs. and no neg. exs. of $\textit{Reach}(x, y)$. FOIL’s measure is 3.20

$\textit{Edge}(x, z)$ is true for all 16 pos. exs. and 4 neg. exs. of $\textit{Reach}(x, y)$. FOIL’s measure is 5.15

Despite FOIL’s judgment, I’ll select:

\[\textit{Reach}(x, y) \leftarrow \textit{Edge}(x, y)\]

for the first rule, leaving the following pos. exs.:

\[
(A, C) \quad (A, D) \quad (A, E) \quad (B, B) \quad (B, D) \quad (B, E) \\
(C, B) \quad (C, C) \quad (C, E) \quad (D, C) \quad (D, D)
\]
Second Rule for Example

Now \( \text{Reach}(x, y) \leftarrow \text{Edge}(x, z) \) is true for all 11 pos. exs. and the following 4 neg. exs. FOIL’s measure is 4.57

\[(A, A) \ (B, A) \ (C, A) \ (D, A)\]

\( \text{Reach}(x, y) \leftarrow \text{Edge}(w, y) \) is also true for all 11 pos. exs., but a different 4 neg. exs.

\[(E, B) \ (E, C) \ (E, D) \ (E, E)\]

Picking \( \text{Edge}(x, z) \), then three further additions are:

\[\text{Reach}(x, y) \leftarrow \text{Edge}(x, z) \land \text{Edge}(z, y)\]

\[\text{Reach}(x, y) \leftarrow \text{Edge}(x, z) \land \text{Edge}(w, y)\]

\[\text{Reach}(x, y) \leftarrow \text{Edge}(x, z) \land \text{Reach}(z, y)\]

All rules are true for 0 neg. exs., but the first rule is true for 5 pos. exs. while the second and third rules are true for all 11.

As a result, the second rule might be:

\[\text{Reach}(x, y) \leftarrow \text{Edge}(x, z) \land \text{Edge}(w, y)\]

This second rule is not very good.

All it requires is an edge from \( x \) and an edge to \( y \).

In fact, FOIL only outputs this one rule on this data.

A graph that has more kinds of paths is needed.

The recursive rule is the correct one.

A special check makes sure infinite recursion is avoided.