Learning Rules

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

if $A_1 \wedge A_2 \wedge \ldots$ **then** C

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

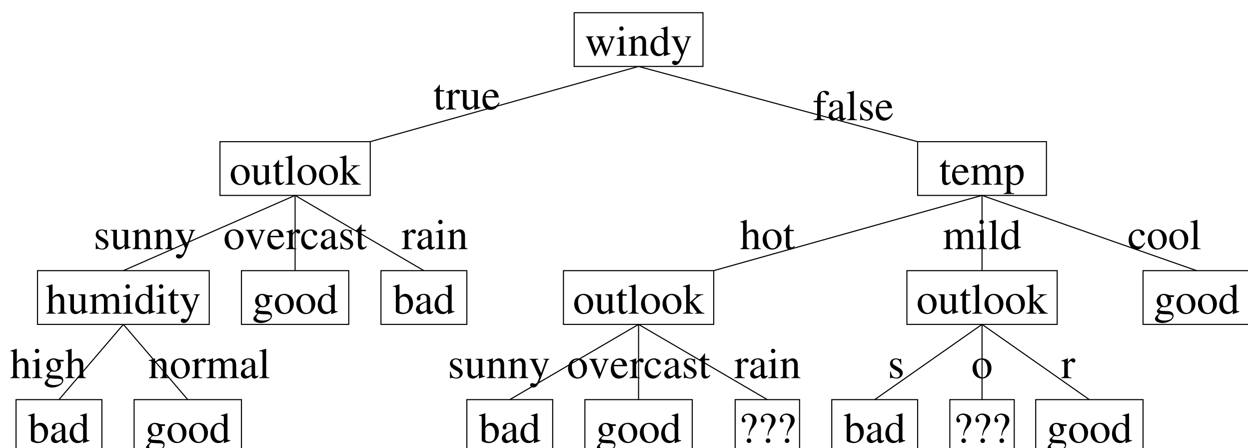
A first-order rule can contain variables, as in:

if $Parent(x, z) \wedge Ancestor(z, y)$ **then** $Ancestor(x, y)$

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:



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

if ¬windy ∧ hot ∧ sunny **then** 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.

if ~~windy~~ $\wedge$ sunny $\wedge$ high **then** bad

if ~~windy~~ $\wedge$ sunny $\wedge$ normal **then** good

if ~~windy~~ $\wedge$ overcast **then** good

if windy $\wedge$ rain **then** bad

if ~~$\neg$ windy~~ $\wedge$ hot $\wedge$ sunny **then** bad

if ~~$\neg$windy $\wedge$ hot~~ $\wedge$ overcast **then** good

if $\neg$windy $\wedge$ mild $\wedge$ sunny **then** bad

if $\neg$windy $\wedge$ ~~mild~~ $\wedge$ rain **then** good

if $\neg$windy $\wedge$ cool **then** good

Rule Ordering

Order rules by accuracy and coverage.

4 **if** overcast **then** good

3 **if** $\neg$windy $\wedge$ rain **then** good

3 **if** sunny $\wedge$ high **then** bad

2 **if** sunny $\wedge$ normal **then** good

2 **if** $\neg$windy $\wedge$ cool **then** good

2 **if** windy $\wedge$ rain **then** bad

2 **if** hot $\wedge$ sunny **then** bad

1 **if** $\neg$windy $\wedge$ mild $\wedge$ 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
  best ← ∅, *frontier* ← {*best*}
  while *frontier* is not empty
    candidates ← all hypotheses generated by adding a
        test to a hypothesis in *frontier*
    best ← best_1 hypothesis in {*best*} ∪ *candidates*
    frontier ← best_2 k hypotheses in *candidates*
  return *best*

"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 $Negative(-3)$, where $Negative$ 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 $Mother(Liz, Charley)$, $Mother$ is a predicate and Liz and $Charley$ are constants.

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

Terminology: Examples of Rules

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

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

Rules can be recursive:
$(x+y=z) \leftarrow (u+v=z) \wedge (x+1=u) \wedge (v+1=y)$
where $x+y=z$ is short for $Equal(plus(x, y), z)$.

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).

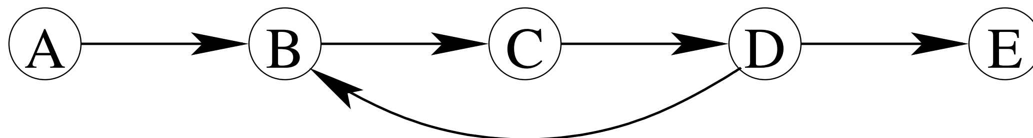
A clause is a disjunction (or) of literals.

A Horn clause has at most one positive literal.

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

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 *Reach* is true for:

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

and false for:

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

We want literals that make $\mathit{Reach}(x,y)$ "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

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

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

Despite FOIL's judgment, I'll select:

$$\mathit{Reach}(x,y) \leftarrow \mathit{Edge}(x,y)$$

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

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

Second Rule for Example

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

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

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

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

Picking $Edge(x, z)$, then three further additions are:

$$Reach(x, y) \leftarrow Edge(x, z) \wedge Edge(z, y)$$

$$Reach(x, y) \leftarrow Edge(x, z) \wedge Edge(w, y)$$

$$Reach(x, y) \leftarrow Edge(x, z) \wedge Reach(z, y)$$

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:

$$Reach(x, y) \leftarrow Edge(x, z) \wedge Edge(w, y)$$

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.