

CS 391L: Machine Learning: Rule Learning

Raymond J. Mooney
University of Texas at Austin

1

Learning Rules

- If-then rules in logic are a standard representation of knowledge that have proven useful in expert-systems and other AI systems
 - In propositional logic a set of rules for a concept is equivalent to DNF
- Rules are fairly easy for people to understand and therefore can help provide insight and comprehensible results for human users.
 - Frequently used in data mining applications where goal is discovering understandable patterns in data.
- Methods for automatically inducing rules from data have been shown to build more accurate expert systems than human knowledge engineering for some applications.
- Rule-learning methods have been extended to first-order logic to handle relational (structural) representations.
 - Inductive Logic Programming (ILP) for learning Prolog programs from I/O pairs.
 - Allows moving beyond simple feature-vector representations of data.

2

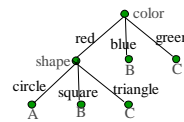
Rule Learning Approaches

- Translate decision trees into rules (C4.5)
- Sequential (set) covering algorithms
 - General-to-specific (top-down) (CN2, FOIL)
 - Specific-to-general (bottom-up) (GOLEM, CIGOL)
 - Hybrid search (AQ, Chillin, Progol)
- Translate neural-nets into rules (TREPAN)

3

Decision-Trees to Rules

- For each path in a decision tree from the root to a leaf, create a rule with the conjunction of tests along the path as an antecedent and the leaf label as the consequent.



red \wedge circle \rightarrow A
 blue \rightarrow B
 red \wedge square \rightarrow B
 green \rightarrow C
 red \wedge triangle \rightarrow C

4

Post-Processing Decision-Tree Rules

- Resulting rules may contain unnecessary antecedents that are not needed to remove negative examples and result in over-fitting.
- Rules are post-pruned by greedily removing antecedents or rules until performance on training data or validation set is significantly harmed.
- Resulting rules may lead to competing conflicting conclusions on some instances.
- Sort rules by training (validation) accuracy to create an ordered decision list. The first rule in the list that applies is used to classify a test instance.

red \wedge circle \rightarrow A (97% train accuracy)
 red \wedge big \rightarrow B (95% train accuracy)
 :
 :
 Test case: <big, red, circle> assigned to class A

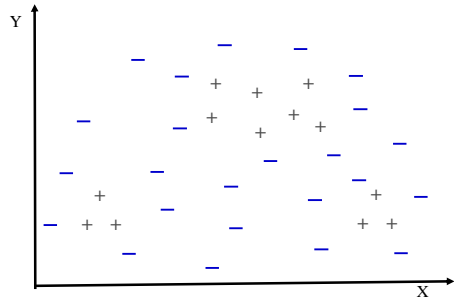
5

Sequential Covering

- A set of rules is learned one at a time, each time finding a single rule that covers a large number of positive instances without covering any negatives, removing the positives that it covers, and learning additional rules to cover the rest.
 - Let P be the set of positive examples
 - Until P is empty do:
 - Learn a rule R that covers a large number of elements of P but no negatives.
 - Add R to the list of rules.
 - Remove positives covered by R from P
- This is an instance of the greedy algorithm for minimum set covering and does not guarantee a minimum number of learned rules.
- Minimum set covering is an NP-hard problem and the greedy algorithm is a standard approximation algorithm.
- Methods for learning individual rules vary.

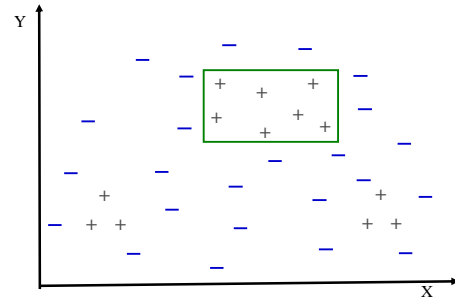
6

Greedy Sequential Covering Example



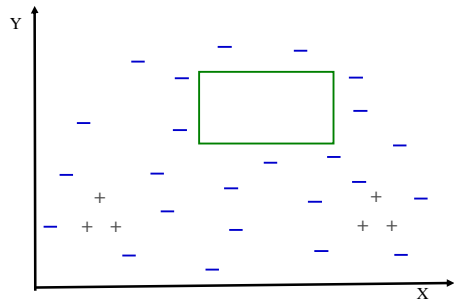
7

Greedy Sequential Covering Example



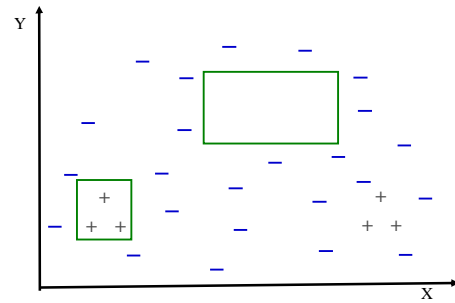
8

Greedy Sequential Covering Example



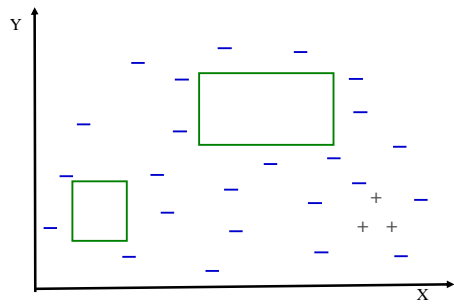
9

Greedy Sequential Covering Example



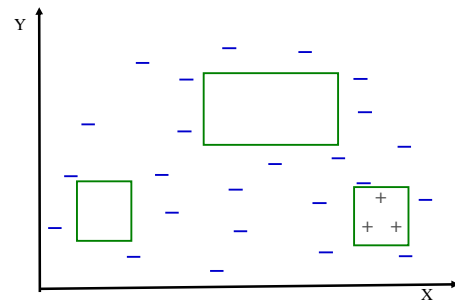
10

Greedy Sequential Covering Example



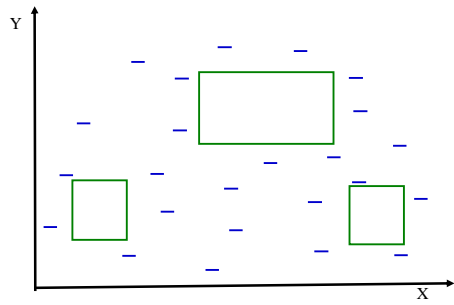
11

Greedy Sequential Covering Example



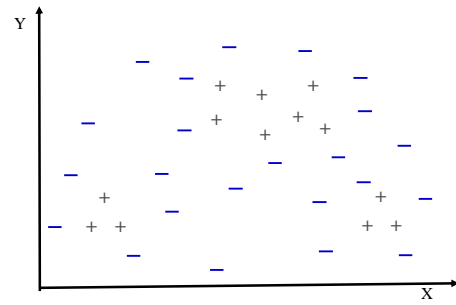
12

Greedy Sequential Covering Example



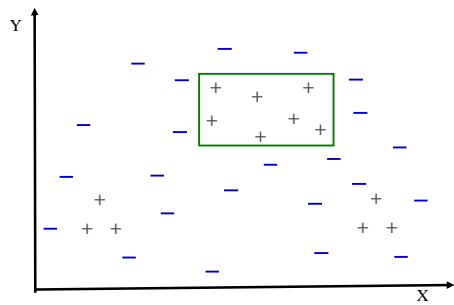
13

No-optimal Covering Example



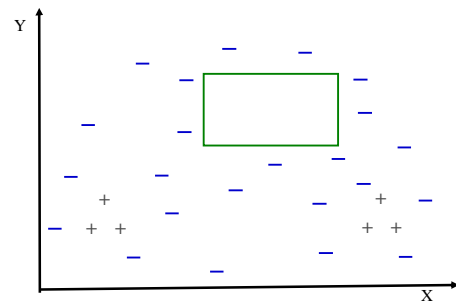
14

Greedy Sequential Covering Example



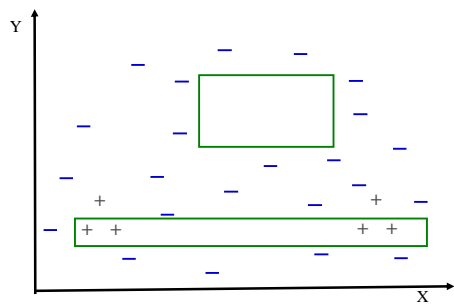
15

Greedy Sequential Covering Example



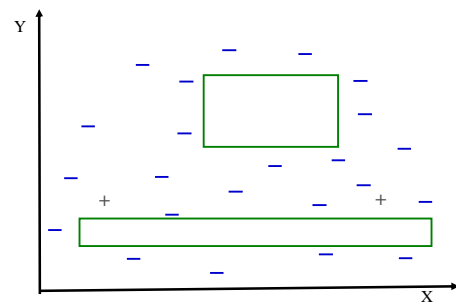
16

Greedy Sequential Covering Example



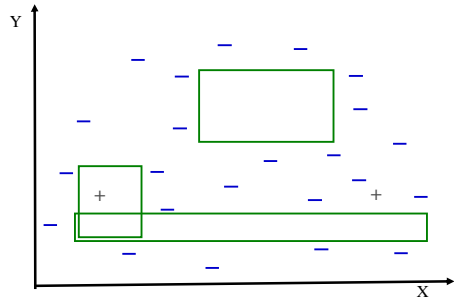
17

Greedy Sequential Covering Example



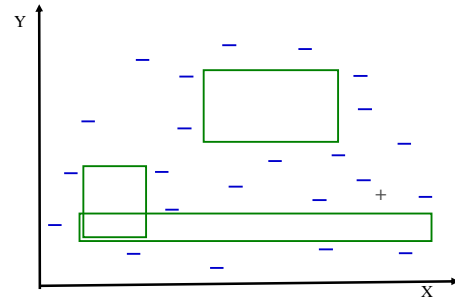
18

Greedy Sequential Covering Example



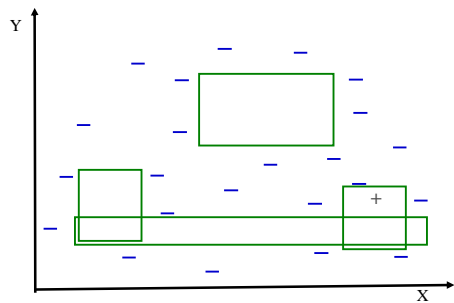
19

Greedy Sequential Covering Example



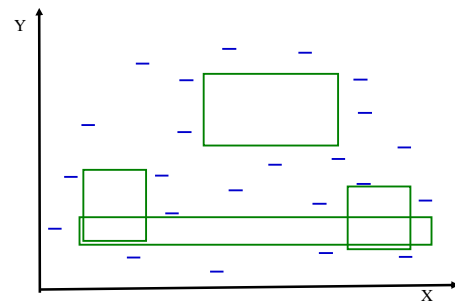
20

Greedy Sequential Covering Example



21

Greedy Sequential Covering Example



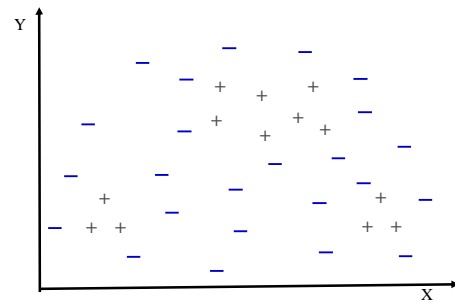
22

Strategies for Learning a Single Rule

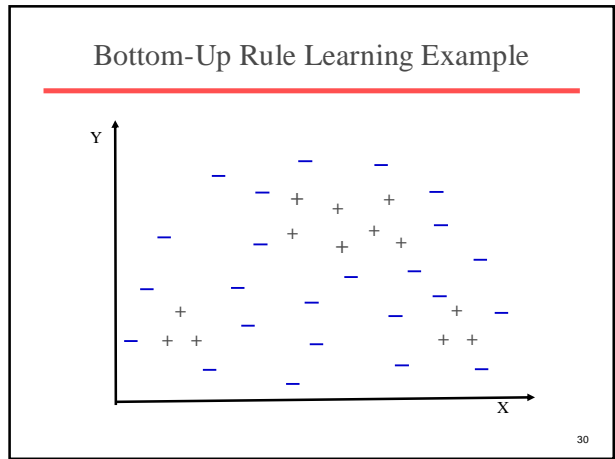
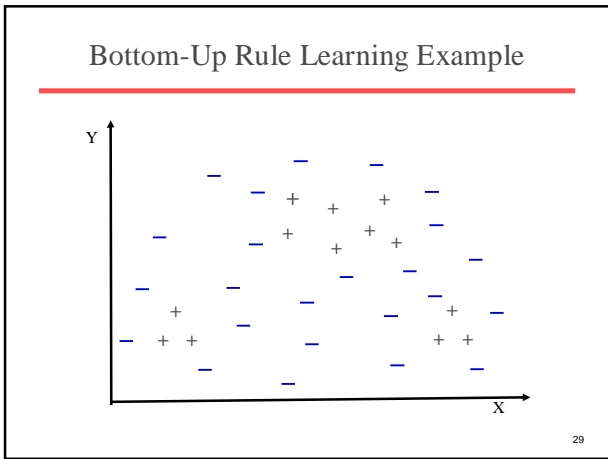
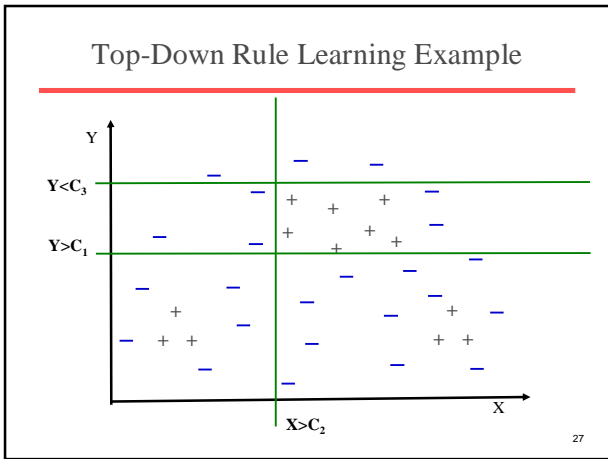
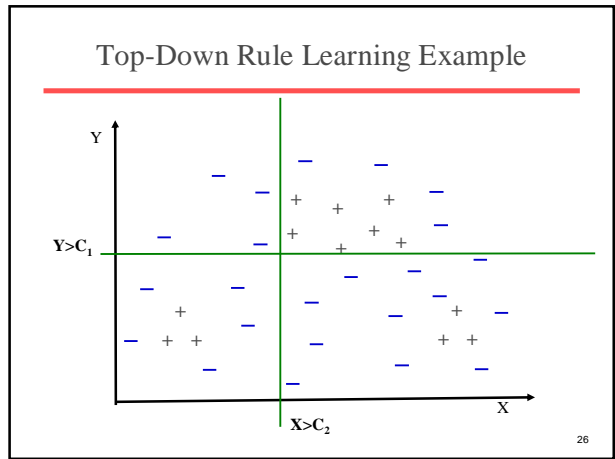
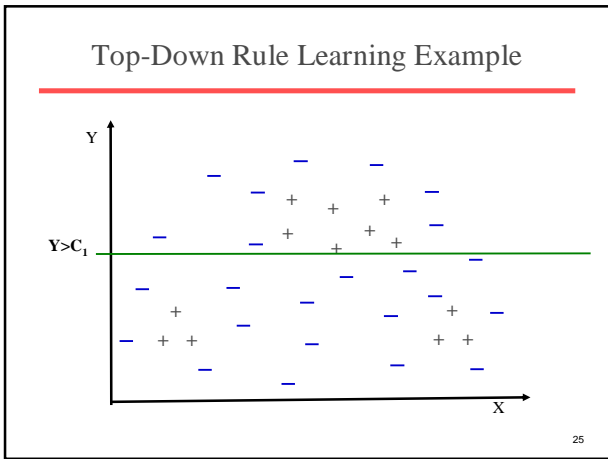
- **Top Down (General to Specific):**
 - Start with the most-general (empty) rule.
 - Repeatedly add antecedent constraints on features that eliminate negative examples while maintaining as many positives as possible.
 - Stop when only positives are covered.
- **Bottom Up (Specific to General)**
 - Start with a most-specific rule (e.g. complete instance description of a random instance).
 - Repeatedly remove antecedent constraints in order to cover more positives.
 - Stop when further generalization results in covering negatives.

23

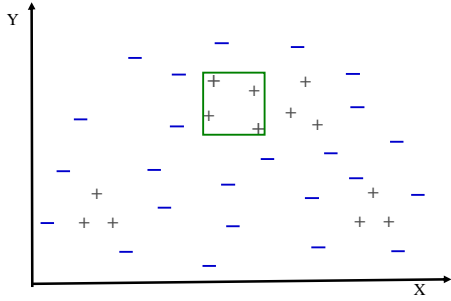
Top-Down Rule Learning Example



24

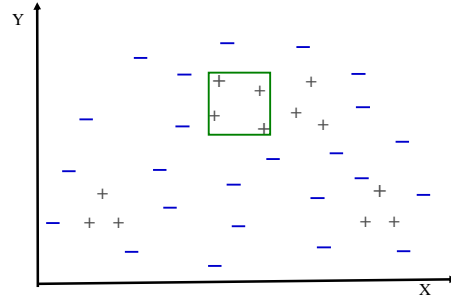


Bottom-Up Rule Learning Example



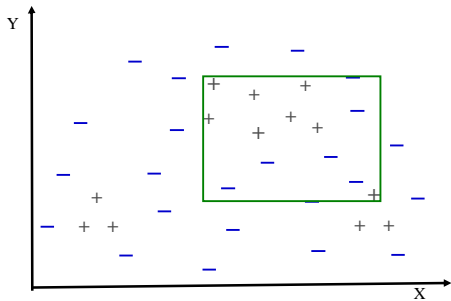
31

Bottom-Up Rule Learning Example



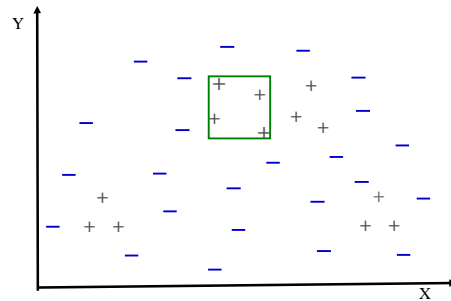
32

Bottom-Up Rule Learning Example



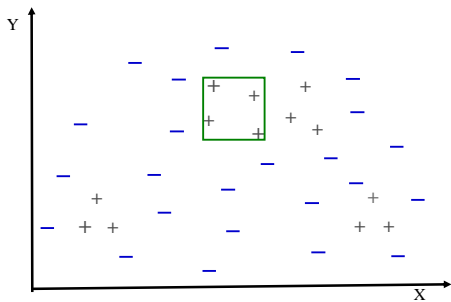
33

Bottom-Up Rule Learning Example



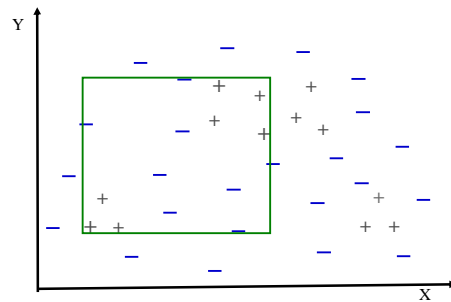
34

Bottom-Up Rule Learning Example



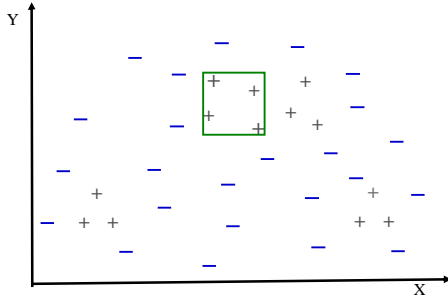
35

Bottom-Up Rule Learning Example



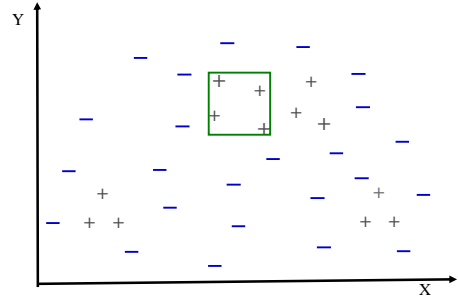
36

Bottom-Up Rule Learning Example



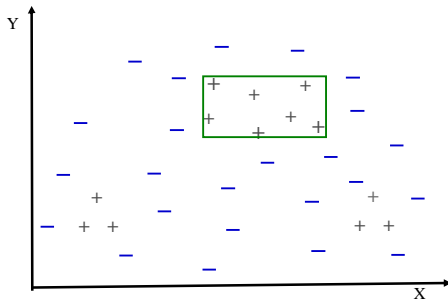
37

Bottom-Up Rule Learning Example



38

Bottom-Up Rule Learning Example



39

Learning a Single Rule in FOIL

- Top-down approach originally applied to first-order logic (Quinlan, 1990).
- Basic algorithm for instances with discrete-valued features:

Let $A = \{\}$ (set of rule antecedents)
 Let N be the set of negative examples
 Let P the current set of uncovered positive examples
 Until N is empty do
 For every feature-value pair (literal) ($F_i = V_{ij}$) calculate
 $\text{Gain}(F_i = V_{ij}, P, N)$
 Pick literal, L , with highest gain.
 Add L to A .
 Remove from N any examples that do not satisfy L .
 Remove from P any examples that do not satisfy L .
 Return the rule: $A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow \text{Positive}$

40

Foil Gain Metric

- Want to achieve two goals
 - Decrease coverage of negative examples
 - Measure increase in percentage of positives covered when literal is added to the rule.
 - Maintain coverage of as many positives as possible.
 - Count number of positives covered.

Define $\text{Gain}(L, P, N)$

Let p be the subset of examples in P that satisfy L .

Let n be the subset of examples in N that satisfy L .

Return: $|p| * [\log_2(p/(p+n)) - \log_2(|P|/(|P|+|N|))]$

41

Sample Disjunctive Learning Data

Example	Size	Color	Shape	Category
1	small	red	circle	positive
2	big	red	circle	positive
3	small	red	triangle	negative
4	big	blue	circle	negative
5	medium	red	circle	negative

42

Propositional FOIL Trace

New Disjunct:
 SIZE=BIG Gain: 0.322
 SIZE=MEDIUM Gain: 0.000
 SIZE=SMALL Gain: 0.322
 COLOR=BLUE Gain: 0.000
 COLOR=RED Gain: 0.644
 COLOR=GREEN Gain: 0.000
 SHAPE=SQUARE Gain: 0.000
 SHAPE=TRIANGLE Gain: 0.000
 SHAPE=CIRCLE Gain: 0.644
 Best feature: COLOR=RED

SIZE=BIG Gain: 1.000
 SIZE=MEDIUM Gain: 0.000
 SIZE=SMALL Gain: 0.000
 SHAPE=SQUARE Gain: 0.000
 SHAPE=TRIANGLE Gain: 0.000
 SHAPE=CIRCLE Gain: 0.830
 Best feature: SIZE=BIG
 Learned Disjunct: COLOR=RED & SIZE=BIG

43

Propositional FOIL Trace

New Disjunct:
 SIZE=BIG Gain: 0.000
 SIZE=MEDIUM Gain: 0.000
 SIZE=SMALL Gain: 1.000
 COLOR=BLUE Gain: 0.000
 COLOR=RED Gain: 0.415
 COLOR=GREEN Gain: 0.000
 SHAPE=SQUARE Gain: 0.000
 SHAPE=TRIANGLE Gain: 0.000
 SHAPE=CIRCLE Gain: 0.415
 Best feature: SIZE=SMALL

COLOR=BLUE Gain: 0.000
 COLOR=RED Gain: 0.000
 COLOR=GREEN Gain: 0.000
 SHAPE=SQUARE Gain: 0.000
 SHAPE=TRIANGLE Gain: 0.000
 SHAPE=CIRCLE Gain: 1.000
 Best feature: SHAPE=CIRCLE
 Learned Disjunct: SIZE=SMALL & SHAPE=CIRCLE
 Final Definition: COLOR=RED & SIZE=BIG v SIZE=SMALL & SHAPE=CIRCLE

44

Rule Pruning in FOIL

- Prepruning method based on *minimum description length* (MDL) principle.
- Postpruning to eliminate unnecessary complexity due to limitations of greedy algorithm.

For each rule, R

For each antecedent, A , of rule
 If deleting A from R does not cause
 negatives to become covered
 then delete A

For each rule, R

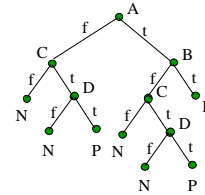
If deleting R does not uncover any positives (since they
 are redundantly covered by other rules)
 then delete R

45

Rule Learning Issues

- Which is better rules or trees?
 - Trees share structure between disjuncts.
 - Rules allow completely independent features in each disjunct.
 - Mapping some rules sets to decision trees results in an exponential increase in size.

$A \wedge B \rightarrow P$
 $C \wedge D \rightarrow P$



What if add rule:
 $E \wedge F \rightarrow P$
 ??

46

Rule Learning Issues

- Which is better top-down or bottom-up search?
 - Bottom-up is more subject to noise, e.g. the random seeds that are chosen may be noisy.
 - Top-down is wasteful when there are many features which do not even occur in the positive examples (e.g. text categorization).

47

Rule Learning vs. Knowledge Engineering

- An influential experiment with an early rule-learning method (AQ) by Michalski (1980) compared results to knowledge engineering (acquiring rules by interviewing experts).
- People known for not being able to articulate their knowledge well.
- Knowledge engineered rules:
 - Weights associated with each feature in a rule
 - Method for summing evidence similar to *certainty factors*.
 - No explicit disjunction
- Data for induction:
 - Examples of 15 soybean plant diseases described using 35 nominal and discrete ordered features, 630 total examples.
 - 290 “best” (diverse) training examples selected for training. Remainder used for testing
 - What is wrong with this methodology?

48

“Soft” Interpretation of Learned Rules

- Certainty of match calculated for each category.
- Scoring method:
 - Literals: 1 if match, -1 if not
 - Terms (conjunctions in antecedent): Average of literal scores.
 - DNF (disjunction of rules): Probabilistic sum: $c_1 + c_2 - c_1 * c_2$
- Sample score for instance $A \wedge B \wedge \neg C \wedge D \wedge \neg E \wedge F$
 - $A \wedge B \wedge C \rightarrow P \quad (1 + 1 + -1)/3 = 0.333$
 - $D \wedge E \wedge F \rightarrow P \quad (1 + -1 + 1)/3 = 0.333$
 - Total score for P: $0.333 + 0.333 - 0.333 * 0.333 = 0.555$
- Threshold of 0.8 certainty to include in possible diagnosis set.

49

Experimental Results

- Rule construction time:
 - Human: 45 hours of expert consultation
 - AQ11: 4.5 minutes training on IBM 360/75
 - What doesn't this account for?
- Test Accuracy:

	1 st choice correct	Some choice correct	Number of diagnoses
AQ11	97.6%	100.0%	2.64
Manual KE	71.8%	96.9%	2.90

50

Relational Learning and Inductive Logic Programming (ILP)

- Fixed feature vectors are a very limited representation of instances.
- Examples or target concept may require relational representation that includes multiple entities with relationships between them (e.g. a graph with labeled edges and nodes).
- First-order predicate logic is a more powerful representation for handling such relational descriptions.
- Horn clauses (i.e. if-then rules in predicate logic, Prolog programs) are a useful restriction on full first-order logic that allows decidable inference.
- Allows learning programs from sample I/O pairs.

51

ILP Examples

- Learn definitions of family relationships given data for primitive types and relations.
 - $\text{uncle}(A,B) :- \text{brother}(A,C), \text{parent}(C,B).$
 - $\text{uncle}(A,B) :- \text{husband}(A,C), \text{sister}(C,D), \text{parent}(D,B).$
- Learn recursive list programs from I/O pairs.
 - $\text{member}(X, [X | _]).$
 - $\text{member}(X, _ | _):- \text{member}(X, _).$
 - $\text{append}([], _, _).$
 - $\text{append}([X | _], _, _):- \text{append}(_, _, _).$

52

ILP

- Goal is to induce a Horn-clause definition for some target predicate P , given definitions of a set of background predicates Q .
- Goal is to find a syntactically simple Horn-clause definition, D , for P given background knowledge B defining the background predicates Q .
 - For every positive example p_i of P
 $D \cup B \models p_i$
 - For every negative example n_i of P
 $D \cup B \not\models n_i$
- Background definitions are provided either:
 - Extensionally: List of ground tuples satisfying the predicate.
 - Intensionally: Prolog definitions of the predicate.

53

ILP Systems

- Top-Down:
 - FOIL (Quinlan, 1990)
- Bottom-Up:
 - CIGOL (Muggleton & Buntine, 1988)
 - GOLEM (Muggleton, 1990)
- Hybrid:
 - CHILLIN (Mooney & Zelle, 1994)
 - PROGOL (Muggleton, 1995)
 - ALEPH (Srinivasan, 2000)

54

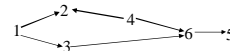
FOIL First-Order Inductive Logic

- Top-down sequential covering algorithm “upgraded” to learn Prolog clauses, but without logical functions.
 - Background knowledge must be provided extensionally.
 - Initialize clause for target predicate P to $P(X_1, \dots, X_T) :-$.
 - Possible specializations of a clause include adding all possible literals:
 - $Q(V_1, \dots, V_T)$
 - $\text{not}(Q(V_1, \dots, V_T))$
 - $X_i = X_j$
 - $\text{not}(X_i = X_j)$
- where X 's are “bound” variables already in the existing clause; at least one of V_1, \dots, V_T is a bound variable, others can be new.
- Allow recursive literals $P(V_1, \dots, V_T)$ if they do not cause an infinite regress.
 - Handle alternative possible values of new intermediate variables by maintaining examples as tuples of all variable values.

55

FOIL Training Data

- For learning a recursive definition, the positive set must consist of *all* tuples of constants that satisfy the target predicate, given some fixed universe of constants.
- Background knowledge consists of complete set of tuples for each background predicate for this universe.
- Example: Consider learning a definition for the target predicate `path` for finding a path in a directed acyclic graph.

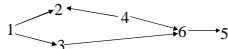


edge: { <1, 2>, <1, 3>, <3, 6>, <4, 2>, <4, 6>, <6, 5> }
 path: { <1, 2>, <1, 3>, <1, 6>, <1, 5>, <3, 6>, <3, 5>, <4, 2>, <4, 6>, <4, 5>, <6, 5> }

56

FOIL Negative Training Data

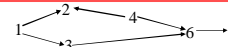
- Negative examples of target predicate can be provided directly, or generated indirectly by making a *closed world assumption*.
 - Every pair of constants $\langle X, Y \rangle$ not in positive tuples for `path` predicate.



Negative path tuples:
 { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

57

Sample FOIL Induction

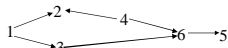


Pos: { <1, 2>, <1, 3>, <1, 6>, <1, 5>, <3, 6>, <3, 5>, <4, 2>, <4, 6>, <4, 5>, <6, 5> }
 Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Start with clause:
`path(X, Y) :-`
 Possible literals to add:
`edge(X, X), edge(Y, Y), edge(X, Y), edge(Y, X), edge(X, Z), edge(Y, Z), edge(Z, X), edge(Z, Y), path(X, X), path(Y, Y), path(X, Y), path(Y, X), path(X, Z), path(Y, Z), path(Z, X), path(Z, Y), X=Y,`
 plus negations of all of these.

58

Sample FOIL Induction

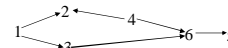


Pos: { <1, 2>, <1, 3>, <1, 6>, <1, 5>, <3, 6>, <3, 5>, <4, 2>, <4, 6>, <4, 5>, <6, 5> }
 Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:
`path(X, Y) :- edge(X, X).`
 Covers 0 positive examples
 Covers 6 negative examples
 Not a good literal.

59

Sample FOIL Induction

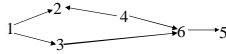


Pos: { <1, 2>, <1, 3>, <1, 6>, <1, 5>, <3, 6>, <3, 5>, <4, 2>, <4, 6>, <4, 5>, <6, 5> }
 Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:
`path(X, Y) :- edge(X, Y).`
 Covers 6 positive examples
 Covers 0 negative examples
 Chosen as best literal. Result is base clause.

60

Sample FOIL Induction



Pos: { <1, 6>, <1, 5>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Y).

Covers 6 positive examples

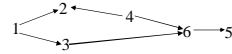
Covers 0 negative examples

Chosen as best literal. Result is base clause.

Remove covered positive tuples.

61

Sample FOIL Induction



Pos: { <1, 6>, <1, 5>, <3, 5>, <4, 5> }

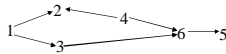
Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Start new clause

path(X, Y) :- .

62

Sample FOIL Induction



Pos: { <1, 6>, <1, 5>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Y).

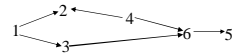
Covers 0 positive examples

Covers 0 negative examples

Not a good literal.

63

Sample FOIL Induction



Pos: { <1, 6>, <1, 5>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>, <2, 6>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>, <5, 6>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

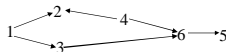
Covers all 4 positive examples

Covers 14 of 26 negative examples

Eventually chosen as best possible literal

64

Sample FOIL Induction



Pos: { <1, 6>, <1, 5>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

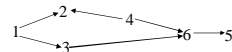
Covers 15 of 26 negative examples

Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

65

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

Covers 15 of 26 negative examples

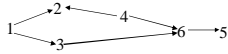
Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

Expand tuples to account for possible Z values.

66

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

Covers 15 of 26 negative examples

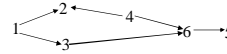
Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

Expand tuples to account for possible Z values.

67

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5> }

Neg: { <1, 1>, <1, 4>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

Covers 15 of 26 negative examples

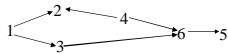
Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

Expand tuples to account for possible Z values.

68

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5, 2>, <4, 5, 6> }

Neg: { <1, 1>, <1, 4>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <4, 1>, <4, 3>, <4, 4>, <6, 1>, <6, 2>, <6, 3>, <6, 4>, <6, 6> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

Covers 15 of 26 negative examples

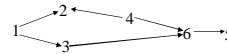
Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

Expand tuples to account for possible Z values.

69

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5, 2>, <4, 5, 6> }

Neg: { <1, 1, 2>, <1, 1, 3>, <1, 4, 2>, <1, 4, 3>, <3, 1, 6>, <3, 2, 6>, <3, 3, 6>, <3, 4, 6>, <4, 1, 2>, <4, 1, 6>, <4, 3, 2>, <4, 3, 6>, <4, 4, 2>, <4, 4, 6>, <6, 1, 5>, <6, 2, 5>, <6, 3, 5>, <6, 4, 5>, <6, 6, 5> }

Test:

path(X, Y) :- edge(X, Z).

Covers all 4 positive examples

Covers 15 of 26 negative examples

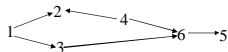
Eventually chosen as best possible literal

Negatives still covered, remove uncovered examples.

Expand tuples to account for possible Z values.

70

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5, 2>, <4, 5, 6> }

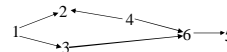
Neg: { <1, 1, 2>, <1, 1, 3>, <1, 4, 2>, <1, 4, 3>, <3, 1, 6>, <3, 2, 6>, <3, 3, 6>, <3, 4, 6>, <4, 1, 2>, <4, 1, 6>, <4, 3, 2>, <4, 3, 6>, <4, 4, 2>, <4, 4, 6>, <6, 1, 5>, <6, 2, 5>, <6, 3, 5>, <6, 4, 5>, <6, 6, 5> }

Continue specializing clause:

path(X, Y) :- edge(X, Z).

71

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5, 2>, <4, 5, 6> }

Neg: { <1, 1, 2>, <1, 1, 3>, <1, 4, 2>, <1, 4, 3>, <3, 1, 6>, <3, 2, 6>, <3, 3, 6>, <3, 4, 6>, <4, 1, 2>, <4, 1, 6>, <4, 3, 2>, <4, 3, 6>, <4, 4, 2>, <4, 4, 6>, <6, 1, 5>, <6, 2, 5>, <6, 3, 5>, <6, 4, 5>, <6, 6, 5> }

Test:

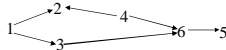
path(X, Y) :- edge(X, Z), edge(Z, Y).

Covers 3 positive examples

Covers 0 negative examples

72

Sample FOIL Induction



Pos: { <1, 6, 2>, <1, 6, 3>, <1, 5, 2>, <1, 5, 3>, <3, 5, 6>, <4, 5, 2>, <4, 5, 6> }

Neg: { <1, 1, 2>, <1, 1, 3>, <1, 4, 2>, <1, 4, 3>, <3, 1, 6>, <3, 2, 6>, <3, 3, 6>, <3, 4, 6>, <4, 1, 2>, <4, 1, 6>, <4, 3, 2>, <4, 3, 6>, <4, 4, 2>, <4, 4, 6>, <6, 1, 5>, <6, 2, 5>, <6, 3, 5>, <6, 4, 5>, <6, 6, 5> }

Test:

path(X, Y) :- edge(X, Z), path(Z, Y).

Covers 4 positive examples

Covers 0 negative examples

Eventually chosen as best literal; completes clause.

Definition complete, since all original <X, Y> tuples are covered (by way of covering some <X, Y, Z> tuple.)

73

Picking the Best Literal

- Same as in propositional case but must account for multiple expanding tuples.

P is the set of positive tuples before adding literal L

N is the set of negative tuples before adding literal L

p is the set of expanded positive tuples after adding literal L

n is the set of expanded negative tuples after adding literal L

$p+$ is the subset of positive tuples before adding L that satisfy L and are expanded into one or more of the resulting set of positive tuples, p .

Return: $|p+| * [\log_2(|p|/(|p+|n|)) - \log_2(|P|/(|P+|N|))]$

- The number of possible literals generated for a predicate is exponential in its arity and grows combinatorially as more new variables are introduced. So the branching factor can be very large.

74

Recursion Limitation

- Must not build a clause that results in an infinite regress.
 - path(X, Y) :- path(X, Y).
 - path(X, Y) :- path(Y, X).
- To guarantee termination of the learned clause, must “reduce” at least one argument according some well-founded partial ordering.
- A binary predicate, R , is a well-founded partial ordering if the transitive closure does not contain $R(a, a)$ for any constant a .
 - rest(A, B)
 - edge(A, B) for an acyclic graph

75

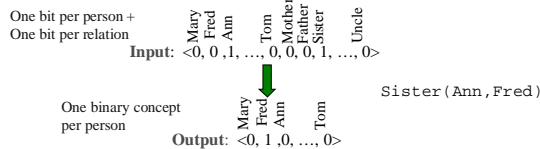
Ensuring Termination in FOIL

- First empirically determines all binary-predicates in the background that form a well-founded partial ordering by computing their transitive closures.
- Only allows recursive calls in which one of the arguments is reduced according to a known well-founded partial ordering.
 - path(X, Y) :- edge(X, Z), path(Z, Y).
 - X is reduced to Z by edge so this recursive call is O.K
- May prevent legal recursive calls that terminate for some other more-complex reason.
- Due to halting problem, cannot determine if an arbitrary recursive definition is guaranteed to halt.

76

Learning Family Relations

- FOIL can learn accurate Prolog definitions of family relations such as wife, husband, mother, father, daughter, son, sister, brother, aunt, uncle, nephew and niece, given basic data on parent, spouse, and gender for a particular family.
- Produces significantly more accurate results than feature-based learners (e.g. neural nets) applied to a “flattened” (“propositionalized”) and restricted version of the problem.



77

Inducing Recursive List Programs

- FOIL can learn simple Prolog programs from I/O pairs.
- In Prolog, lists are represented using a logical function cons(Head, Tail) written as [Head | Tail].
- Since FOIL cannot handle functions, this is re-represented as a predicate:
 - components(List, Head, Tail)
- In general, an m -ary function can be replaced by a $(m+1)$ -ary predicate.

78

Example: Learn Prolog Program for List Membership

- **Target:**
 - member:
(a,[a]),(b,[b]),(a,[a,b]),(b,[a,b]),...
- **Background:**
 - components:
([a],a,[]),([b],b,[]),([a,b],a,[b]),
([b,a],b,[a]),([a,b,c],a,[b,c]),...
- **Definition:**
 - member(A,B) :- components(B,A,C).
 - member(A,B) :- components(B,C,D),
member(A,D).

79

Logic Program Induction in FOIL

- FOIL has also learned
 - append given components and null
 - reverse given append, components, and null
 - quicksort given partition, append, components, and null
 - Other programs from the first few chapters of a Prolog text.
- **Learning recursive programs in FOIL requires a complete set of positive examples for some constrained universe of constants, so that a recursive call can always be evaluated extensionally.**
 - For lists, all lists of a limited length composed from a small set of constants (e.g. all lists up to length 3 using {a,b,c}).
 - Size of extensional background grows combinatorially.
- **Negative examples usually computed using a closed-world assumption.**
 - Grows combinatorially large for higher arity target predicates.
 - Can randomly sample negatives to make tractable.

80

More Realistic Applications

- **Classifying chemical compounds as mutagenic (cancer causing) based on their graphical molecular structure and chemical background knowledge.**
- **Classifying web documents based on both the content of the page and its links to and from other pages with particular content.**
 - A web page is a university faculty home page if:
 - It contains the words “Professor” and “University”, and
 - It is pointed to by a page with the word “faculty”, and
 - It points to a page with the words “course” and “exam”

81

FOIL Limitations

- **Search space of literals (branching factor) can become intractable.**
 - Use aspects of bottom-up search to limit search.
- **Requires large extensional background definitions.**
 - Use intensional background via Prolog inference.
- **Hill-climbing search gets stuck at local optima and may not even find a consistent clause.**
 - Use limited backtracking (beam search)
 - Include determinate literals with zero gain.
 - Use relational pathfinding or relational clichés.
- **Requires complete examples to learn recursive definitions.**
 - Use intensional interpretation of learned recursive clauses.

82

FOIL Limitations (cont.)

- **Requires a large set of closed-world negatives.**
 - Exploit “output completeness” to provide “implicit” negatives.
 - past-tense([s,i,n,g], [s,a,n,g])
- **Inability to handle logical functions.**
 - Use bottom-up methods that handle functions
- **Background predicates must be sufficient to construct definition, e.g. cannot learn reverse unless given append.**
 - Predicate invention
 - Learn reverse by inventing append
 - Learn sort by inventing insert

83

Rule Learning and ILP Summary

- There are effective methods for learning symbolic rules from data using greedy sequential covering and top-down or bottom-up search.
- These methods have been extended to first-order logic to learn relational rules and recursive Prolog programs.
- Knowledge represented by rules is generally more interpretable by people, allowing human insight into what is learned and possible human approval and correction of learned knowledge.

84