Data Mining

Lesson 4

Rule Induction

MSc in Computer Science
University of New York Tirana
Assoc. Prof. Dr. Marenglen Biba

Data Mining: Content

- Introduction to data mining and machine learning
- Inductive learning
- Decision trees
- Rule induction
- Instance-based learning
- Bayesian learning
- Neural networks
- Support vector machines
- Other machine learning models
- Engineering data mining tasks

Rule Induction

- Theory
  – Propositional Rule Learning
  – First-Order Rule Learning
  – FOIL

- Practice
  – Rule Induction in Weka
  – Experiments on Mutagenesis with the INTHELEX system

Introduction

- In many cases it is useful to learn the target function represented as a set of if-then rules that jointly define the function.
- In this lesson we explore a variety of algorithms that directly learn rule sets and that differ from these algorithms in two key respects.
- First, they are designed to learn sets of first-order rules that contain variables.
  - This is significant because first-order rules are much more expressive than propositional rules.
- Second, the algorithms discussed here use sequential covering algorithms that learn one rule at a time to incrementally grow the final set of rules.

First-order rules

- As an example of first-order rule sets, consider the following two rules that jointly describe the target concept Ancestor.

- Here we use the predicate Parent (x, y) to indicate that y is the mother or father of x, and the predicate Ancestor(x, y) to indicate that y is an ancestor of x related by an arbitrary number of family generations.
  - IF Parent(x,y) THEN Ancestor(x,y)
  - IF Parent (x, z) And Ancestor(z, y) THEN Ancestor (x, y)

Example: Describing protein folds

- 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.

A positive and a negative example of the protein fold "4-helical-up-and-down-bundle".

Positive	Negative

3-D arrangement of secondary structure units shown for -helices (cylinders) and -sheets (arrows).

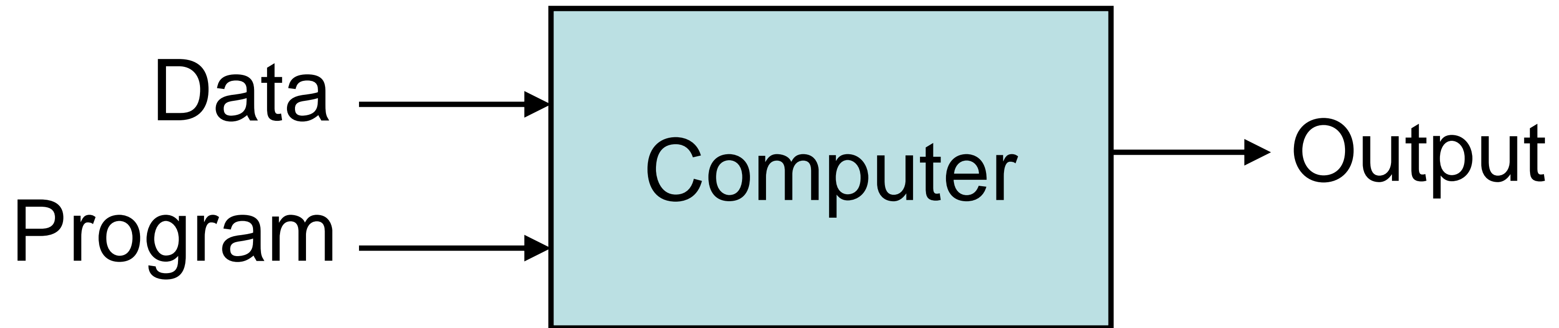
Prolog

- Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.
- Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules.
- A computation is initiated by running a query over these relations.
- Prolog Programming for Artificial Intelligence, I. Bratko.
  - http://www.amazon.com/Programming-Artificial-Intelligence-International-Computer/dp/0321417461

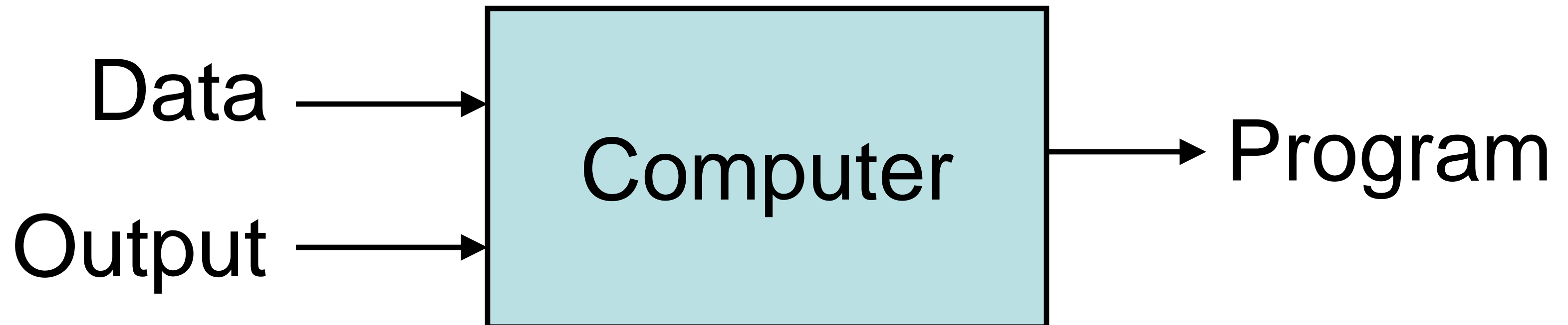
Learning Prolog Programs

- One way to see the representational power of first-order rules is to consider the general purpose programming language Prolog.
- In Prolog, programs are sets of first-order rules such as the two shown previously (rules of this form are also called *Horn clauses*).
- In fact, when stated in a slightly different syntax the above rules form a valid Prolog program for computing the Ancestor relation.
- In this light, a general purpose algorithm capable of learning such rule sets may be viewed as an algorithm for automatically inferring Prolog programs from examples.
- In this lesson we explore learning algorithms capable of learning such rules, given appropriate sets of training examples.

Traditional Programming



Machine Learning



From P. Domingos, CSD-UW

Applications

- In practice, learning systems based on first-order representations have been successfully applied to problems such as:
  - learning which chemical bonds fragment in a mass spectrometer (Buchanan 1976; Lindsay 1980),
  - learning which chemical substructures produce mutagenic activity (a property related to carcinogenicity)
    - Paper by (Srinivasan et al. 1994)
  - discovering pharmacophores
    - Paper (Fin. et al. 1999)

Learning Propositional Rules

Sequential covering algorithms

- Here we consider a family of algorithms for learning rule sets based on the strategy of:
  - learning one rule
  - removing the data it covers,
  - then iterating this process.

- Such algorithms are called sequential covering algorithms.

Learn-one-rule

- To elaborate, imagine we have a subroutine Learn-one-rule that accepts a set of positive and negative training examples as input, then outputs a single rule that covers many of the positive examples and few of the negative examples.
- We require that this output rule have high accuracy, but not necessarily high coverage.
  - By high accuracy, we mean the predictions it makes should be correct.
  - By accepting low coverage, we mean it need not make predictions for every training example.

Sequential covering algorithms

SEQUENTIAL-COVERING(*Target_attribute*, *Attributes*, *Examples*, *Threshold*)

- *Learned_rules* ← {}
- *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- while PERFORMANCE(*Rule*, *Examples*) > *Threshold*, do
    - *Learned_rules* ← *Learned_rules* + *Rule*
    - *Examples* ← *Examples* − {examples correctly classified by *Rule*}
    - *Rule* ← LEARN-ONE-RULE(*Target_attribute*, *Attributes*, *Examples*)
- *Learned_rules* ← sort *Learned_rules* accord to PERFORMANCE over *Examples*
- return *Learned_rules*

TABLE 10.1
The sequential covering algorithm for learning a disjunctive set of rules. LEARN-ONE-RULE must return a single rule that covers at least some of the *Examples*. PERFORMANCE is a user-provided subroutine to evaluate rule quality. This covering algorithm learns rules until it can no longer learn a rule whose performance is above the given *Threshold*.

The algorithm learns a disjunctive set of rules.

General to Specific Search

- One effective approach to implementing Learn-one-rule is to organize the hypothesis space search in the same general fashion as the ID3 algorithm, but to follow only the most promising branch in the tree at each step.

- The search begins by considering the most general rule precondition possible (the empty test that matches every instance), then greedily adding the attribute test that most improves rule performance measured over the training examples.

- Once this test has been added, the process is repeated by greedily adding a second attribute test, and so on.

General to Specific Search

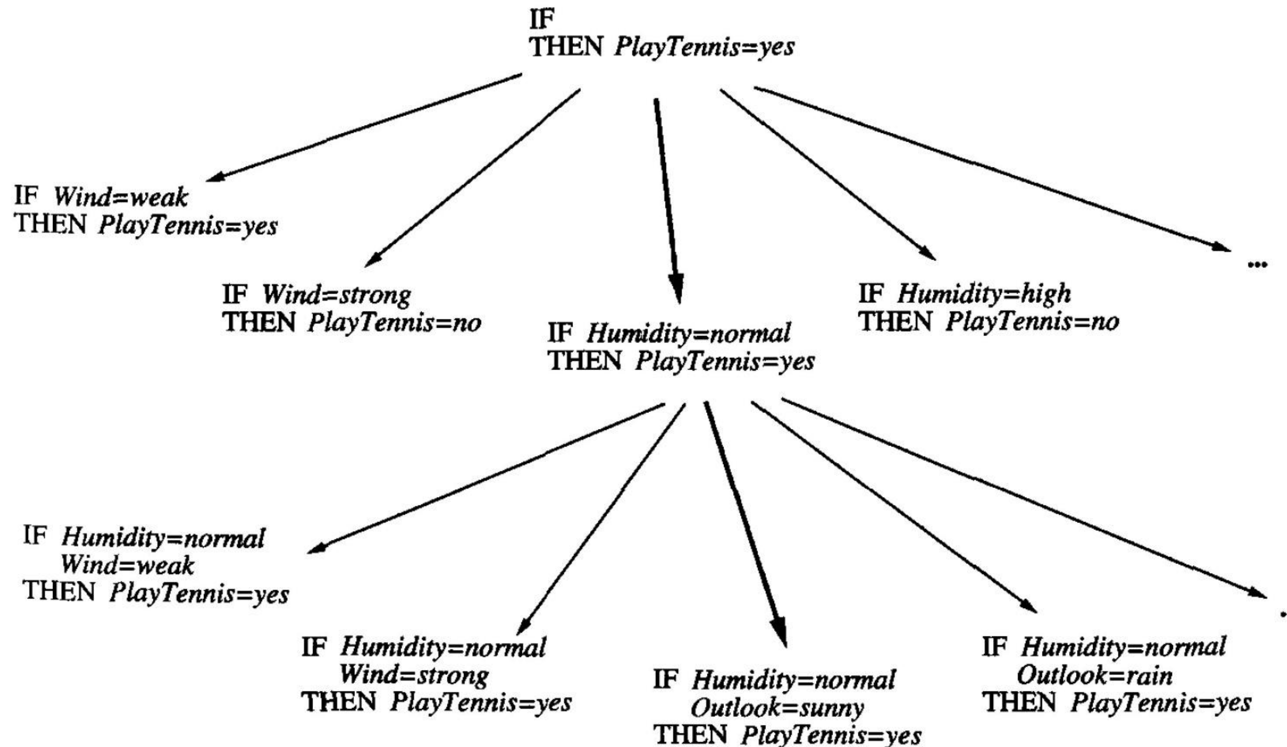


FIGURE 10.1
The search for rule preconditions as LEARN-ONE-RULE proceeds from general to specific. At each step, the preconditions of the best rule are specialized in all possible ways. Rule postconditions are determined by the examples found to satisfy the preconditions. This figure illustrates a beam search of width 1.

Beam Search

- The general-to-specific search suggested above for the Learn-one-rule algorithm is a greedy depth-first search with no backtracking.

- As with any greedy search, there is a danger that a suboptimal choice will be made at any step.

- To reduce this risk, we can extend the algorithm to perform a beam search;
  - that is, a search in which the algorithm maintains a list of the k best candidates at each step, rather than a single best candidate.

Beam Search

- On each search step, descendants (specializations) are generated for each of these k best candidates, and the resulting set is again reduced to the k most promising members.
- Beam search keeps track of the most promising alternatives to the current top-rated hypothesis, so that all of their successors can be considered at each search step.

Issues in Learning Rule Sets

- The Sequential-covering algorithm described above and the decision tree learning algorithms suggest a variety of possible methods for learning sets of rules.
- This section considers several key dimensions in the design space of such rule learning algorithms.

Sequential Vs. Simultaneous

- First, sequential covering algorithms learn one rule at a time, removing the covered examples and repeating the process on the remaining examples.
- In contrast, decision tree algorithms such as ID3 learn the entire set of disjuncts simultaneously as part of the single search for an acceptable decision tree.
- We might, therefore, call algorithms such as ID3 simultaneous covering algorithms, in contrast to sequential covering algorithms such as CN2.
- Which should we prefer?

Sequential Vs. Simultaneous

- Sequential covering algorithms such as CN2 make a larger number of independent choices than simultaneous covering algorithms such as ID3.
- Still, the question remains, which should we prefer?
- The answer may depend on how much training data is available.
- If data is plentiful, then it may support the larger number of independent decisions required by the sequential covering algorithm.
- If data is scarce, the "sharing" of decisions regarding preconditions of different rules may be more effective.

Direction of Search

- A second dimension along which approaches vary is the direction of the search in Learn-one-rule.
- In the algorithm described above, the search is from general to specific hypotheses. Other algorithms we have discussed (e.g., Find-S from Lesson 2) search from specific to general.
- One advantage of general to specific search in this case is that there is a single maximally general hypothesis from which to begin the search, whereas there are very many specific hypotheses in most hypothesis spaces (i.e., one for each possible instance).
- Given many maximally specific hypotheses, it is unclear which to select as the starting point of the search.

Direction of Search

- One program that conducts a specific-to-general search, called Golem (Muggleton and Feng 1990), addresses this issue by choosing several positive examples at random to initialize and to guide the search.

- The best hypothesis obtained through multiple random choices is then selected.

Generate-and-test Vs. Example driven

- A third dimension is whether the Learn-one-rule search is a generate then test search through the syntactically legal hypotheses, as it is in our suggested implementation, or whether it is example-driven so that individual training examples constrain the generation of hypotheses.
- Prototypical example-driven search algorithms include the Find-S and Candidate-Elimination algorithms of Lesson 2.
- In each of these algorithms, the generation or revision of hypotheses is driven by the analysis of an individual training example, and the result is a revised hypothesis designed to correct performance for this single example.
- This contrasts to the generate and test search of Learn-one-rule, in which successor hypotheses are generated based only on the syntax of the hypothesis representation.
  - The training data is considered only after these candidate hypotheses are generated and is used to choose among the candidates based on their performance over the entire collection of training examples.

Generate-and-test Vs. Example driven

- One important advantage of the generate-and-test approach is that each choice in the search is based on the hypothesis performance over many examples, so that the impact of noisy data is minimized.

- In contrast, example-driven algorithms that refine the hypothesis based on individual examples are more easily misled by a single noisy training example and are therefore less robust to errors in the training data.

Post-pruning of rules

- A fourth dimension is whether and how rules are post-pruned.
- As in decision tree learning, it is possible for Learn-one-rule to formulate rules that perform very well on the training data, but less well on subsequent data.
- As in decision tree learning, one way to address this issue is to post-prune each rule after it is learned from the training data.
- In particular, *preconditions can be removed from the rule whenever this leads to improved performance over a set of pruning examples distinct from the training examples.*

Performance: Relative frequency

- A final dimension is the particular definition of rule performance used to guide the search in Learn-one-rule.
- Various evaluation functions have been used. Some common evaluation functions include:

1. Relative frequency. Let *n* denote the number of examples the rule matches and let n_c denote the number of these that it classifies correctly. The relative frequency estimate of rule performance is

$$n_c / n$$

Performance: Entropy

2. Entropy.

- Let S be the set of examples that match the rule preconditions.
- Entropy measures the uniformity of the target function values for this set of examples.
- We take the negative of the entropy so that better rules will have higher scores:

$$-Entropy(S) = \sum_{i=1}^{c} p_i \log_2 p_i$$

- where c is the number of distinct values the target function may take on, and where p_i is the proportion of examples from S for which the target function takes on the *i*th value.
- It is also the basis for the information gain measure used by many decision tree learning algorithms.

Learning First-Order Rules

Learning FO rules

- In this section, we consider learning rules that contain variables — in particular, learning first-order Horn theories.
- Our motivation for considering such rules is that they are much more expressive than propositional rules.
- Inductive learning of first-order rules or theories is often referred to as inductive logic programming (or ILP for short), because this process can be viewed as automatically inferring Prolog programs from examples.
- Prolog is a general purpose, Turing-equivalent programming language in which programs are expressed as collections of Horn clauses.

First-Order Horn Clauses

- To see the advantages of first-order representations over propositional (variable-free) representations, consider the task of learning the simple target concept Daughter (x, y), defined over pairs of people x and y.
- The value of Daughter(x, y) is True when x is the daughter of y, and False otherwise.
- Suppose each person in the data is described by the attributes Name, Mother, Father, Male, Female.
- Hence, each training example will consist of the description of two people in terms of these attributes, along with the value of the target attribute Daughter.
- For example, the following is a positive example in which Sharon is the daughter of Bob:
- (Name1 = Sharon, Mother1 = Louise, Father1 = Bob,
- Male1 = False, Female1 = True,
- Name2 = Bob, Mother2 = Nora, Father2 = Victor,
- Male2 = True, Female2 = False, Daughter1,2 = True)

First-Order Horn Clauses

- Now if we were to collect a number of such training examples for the target concept $Daughter_{1,2}$ and provide them to a propositional rule learner such as CN2 or C4.5, the result would be a collection of very specific rules such as:

  – IF ($Father_1$ = Bob) and ($Name_2$ = Bob) and ($Female_1$ = True) THEN $Daughter_{1,2}$ = True

- Although it is correct, this rule is so specific that it will rarely, if ever, be useful in classifying future pairs of people.

The problem with propositional representations

- *The problem is that propositional representations offer no general way to describe the essential relations among the values of the attributes*.

- In contrast, a program using first-order representations could learn the following general rule:

  IF Father(y,x) AND Female(y), THEN Daughter(x, y)

- where x and y are variables that can be bound to any person.

The problem with propositional representations

- First-order Horn clauses may also refer to variables in the preconditions that do not occur in the postconditions.

- For example, one rule for GrandDaughter might be:

  IF Father(y, z) AND Mother(z, x) AND Female(y)

  THEN GrandDaughter(x, y)

Terminology

- Before moving on to algorithms for learning sets of Horn clauses, let us introduce some basic terminology from formal logic.
- All expressions are composed of constants (e.g., Bob, Louise), variables (e.g., x, y), predicate symbols (e.g., Married, Greater-Than), and function symbols (e.g., age).
- The difference between predicates and functions is that predicates take on values of True or False, whereas functions may take on any constant as their value.
- We will use lowercase symbols for variables and capitalized symbols for constants.

Terminology

- Also, we will use lowercase for functions and capitalized symbols for predicates.
- From these symbols, we build up expressions as follows: A term is any constant, any variable, or any function applied to any term (e.g., Bob, x, age(Bob)).
- A literal is any predicate or its negation applied to any term (e.g., Married(Bob, Louise), ¬Greater_Than(age{Sue), 20)).
- If a literal contains a negation (¬) symbol, we call it a negative literal, otherwise a positive literal.

Terminology

- A clause is any disjunction of literals, where all variables are assumed to be universally quantified.
- A Horn clause is a clause containing at most one positive literal, such as:

$$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- where H is the positive literal, and $\neg L_1 \vee \ldots \vee \neg L_n$ are negative literals.
- Because of the equalities $(B \vee \neg A) = (B \leftarrow A)$ and $\neg(A \wedge B) = (\neg A \vee \neg B)$, the above Horn clause can alternatively be written in the form

$$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

- which is equivalent to the following, using our earlier rule notation

$$\text{IF } L_1 \wedge \ldots \wedge L_n \text{ THEN } H$$

- Whatever the notation, the Horn clause preconditions $L_1 \wedge \ldots \wedge L_n$ are called the clause body or, alternatively, the clause antecedents.
- The literal H that forms the postcondition is called the clause head or, alternatively, the clause consequent.

First-order Logic

- Every well-formed expression is composed of *constants* (e.g., *Mary*, 23, or *Joe*), *variables* (e.g., x), *predicates* (e.g., *Female*, as in *Female*(*Mary*)), and *functions* (e.g., *age*, as in *age*(*Mary*)).
- A *term* is any constant, any variable, or any function applied to any term. Examples include *Mary*, x, *age*(*Mary*), *age*(x).
- A *literal* is any predicate (or its negation) applied to any set of terms. Examples include *Female*(*Mary*), $\neg$*Female*(x), *Greater_than*(*age*(*Mary*), 20).
- A *ground literal* is a literal that does not contain any variables (e.g., $\neg$*Female*(*Joe*)).
- A *negative literal* is a literal containing a negated predicate (e.g., $\neg$*Female*(*Joe*)).
- A *positive literal* is a literal with no negation sign (e.g., *Female*(*Mary*)).
- A *clause* is any disjunction of literals $M_1 \vee \ldots M_n$ whose variables are universally quantified.
- A *Horn clause* is an expression of the form

  $$H \leftarrow (L_1 \wedge \ldots \wedge L_n)$$

  where $H, L_1 \ldots L_n$ are positive literals. H is called the *head* or *consequent* of the Horn clause. The conjunction of literals $L_1 \wedge L_2 \wedge \ldots \wedge L_n$ is called the *body* or *antecedents* of the Horn clause.
- For any literals A and B, the expression $(A \leftarrow B)$ is equivalent to $(A \vee \neg B)$, and the expression $\neg(A \wedge B)$ is equivalent to $(\neg A \vee \neg B)$. Therefore, a Horn clause can equivalently be written as the disjunction

  $$H \vee \neg L_1 \vee \ldots \vee \neg L_n$$

- A *substitution* is any function that replaces variables by terms. For example, the substitution $\{x/3, y/z\}$ replaces the variable x by the term 3 and replaces the variable y by the term z. Given a substitution θ and a literal L we write $L\theta$ to denote the result of applying substitution θ to L.
- A *unifying substitution* for two literals L_1 and L_2 is any substitution θ such that $L_1\theta = L_2\theta$.

TABLE 10.3
Basic definitions from first-order logic.

Learning Sets Of First-Order Rules: FOIL

- A variety of algorithms has been proposed for learning first-order rules, or Horn clauses.

- In this section we consider a program called FOIL, (Quinlan 1990) that employs an approach very similar to the Sequential-covering and Learn-one-Rule algorithms of the previous section.

- In fact, the FOIL program is the natural extension of these earlier algorithms to first-order representations.

FOIL

- Formally, the hypotheses learned by FOIL are sets of first-order rules, where each rule is similar to a Horn clause with two exceptions.
- First, the rules learned by FOIL are more restricted than general Horn clauses, because the literals are not permitted to contain function symbols (this reduces the complexity of the hypothesis space search).
- Second, FOIL rules are more expressive than Horn clauses, because the literals appearing in the body of the rule may be negated.
- FOIL has been applied to a variety of problem domains.
- For example, it has been demonstrated to learn a recursive definition of the Quicksort algorithm and to learn to discriminate legal from illegal chess positions.

The Basic FOIL Algorithm

FOIL(*Target_predicate*, *Predicates*, *Examples*)

- *Pos* ← those *Examples* for which the *Target_predicate* is *True*
- *Neg* ← those *Examples* for which the *Target_predicate* is *False*
- *Learned_rules* ← {}
- while *Pos*, do

  Learn a NewRule
  - *NewRule* ← the rule that predicts *Target_predicate* with no preconditions
  - *NewRuleNeg* ← *Neg*
  - while *NewRuleNeg*, do

    Add a new literal to specialize NewRule
    - *Candidate_literals* ← generate candidate new literals for *NewRule*, based on *Predicates*
    - *Best_literal* ← $\underset{L \in Candidate_literals}{\text{argmax}}$ *Foil_Gain*(*L*, *NewRule*)
    - add *Best_literal* to preconditions of *NewRule*
    - *NewRuleNeg* ← subset of *NewRuleNeg* that satisfies *NewRule* preconditions
  - *Learned_rules* ← *Learned_rules* + *NewRule*
  - *Pos* ← *Pos* − {members of *Pos* covered by *NewRule*}
- Return *Learned_rules*

TABLE 10.4
The basic FOIL algorithm. The specific method for generating *Candidate_literals* and the definition of *Foil_Gain* are given in the text. This basic algorithm can be modified slightly to better accommodate noisy data, as described in the text.

Search in FOIL: two types of search

- The hypothesis space search performed by FOIL is best understood by viewing it hierarchically.
- Each iteration through FOIL'S outer loop adds a new rule to its disjunctive hypothesis, Learned-rules.
- The effect of each new rule is to generalize the current disjunctive hypothesis (i.e., to increase the number of instances it classifies as positive), by adding a new disjunct.
- Viewed at this level, the search is a specific-to-general search through the space of hypotheses, beginning with the most specific empty disjunction and terminating when the hypothesis is sufficiently general to cover all positive training examples.

Search in FOIL

- The inner loop of FOIL performs a finer-grained search to determine the exact definition of each new rule.
- This inner loop searches a second hypothesis space, consisting of conjunctions of literals, to find a conjunction that will form the preconditions for the new rule.
- Within this hypothesis space, it conducts a general-to-specific, hill-climbing search, beginning with the most general preconditions possible (the empty precondition), then adding literals one at a time to specialize the rule until it avoids all negative examples.

FOIL and Learn-one-rule

- The two most substantial differences between FOIL and the earlier Sequential-covering and Learn-one-rule algorithm are:

1. In its general-to-specific search to learn each new rule, FOIL employs different detailed steps to generate candidate specializations of the rule.
   - This difference follows from the need to accommodate variables in the rule preconditions.
2. FOIL employs a performance measure, Foil-Gain, that differs from the entropy measure shown for Learn-one-rule.
   - This difference follows from the need to distinguish between different bindings of the rule variables and from the fact that FOIL seeks only rules that cover positive examples.

Guiding the Search in FOIL

- To select the most promising literal from the candidates generated at each step, FOIL considers the performance of the rule over the training data.

- In doing this, it considers all possible bindings of each variable in the current rule.
  - Ex. notation {x/Bob, y/Sharon} to denote a particular variable binding; that is, a substitution mapping each variable to a constant.

Guiding the Search in FOIL

- At each stage, the rule is evaluated based on the sets of positive and negative variable bindings, with preference given to rules that possess more positive bindings and fewer negative bindings.

- As new literals are added to the rule, the sets of bindings will change.

Evaluation function used by FOIL

- The evaluation function used by FOIL to estimate the utility of adding a new literal is based on the numbers of positive and negative bindings covered before and after adding the new literal.

- More precisely, consider some rule R, and a candidate literal L that might be added to the body of R. Let R_f be the rule created by adding literal L to rule R.

- The value FoilGain(L, R) of adding L to R is defined as:

Information Gain in FOIL

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

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 $R + L$
- $t =$ no. of positive bindings of R also covered by $R + L$

Interpreting FoilGain

- This FoilGain function has a straightforward interpretation in terms of information theory.
- According to information theory, $-\log_2 p_0/(p_0+n_0)$ is the minimum number of bits needed to encode the classification of an arbitrary positive binding among the bindings covered by rule R.
- Similarly, $-\log_2 p_1/(P_1+ n_1)$ is the number of bits required if the binding is one of those covered by rule R.
- Since t is just the number of positive bindings covered by R that remain covered by R', FoilGain(L, R) can be seen as the reduction due to L in the total number of bits needed to encode the classification of all positive bindings of R.

Lab Session

- Rule Induction in Weka
- INTHELEX system
  - Mutagenesis experiments

JRip

- JRip (RIPPER)
- JRip implements a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER), which was proposed by William W. Cohen as an optimized version of IREP. Ripper builds a ruleset by repeatedly adding rules to an empty ruleset until all positive examples are covered.
- Rules are formed by greedily adding conditions to the antecedent of a rule (starting with empty antecendent) until no negative examples are covered. After a ruleset is constructed, an optimization postpass massages the ruleset so as to reduce its size and improve its fit to the training data. A combination of cross-validation and minimum-description length techniques is used to prevent overfitting.
- Cohen, W. W. 1995. Fast effective rule induction. In Machine Learning: Proceedings of the Twelfth International Conference, Lake Tahoe, California.

Examples in Weka

- Iris problem
- Colic
- Credit card approval
- Diabetes
- Glass Identification
- Heart Disease
- Hepatitis
- Thyroid Disease
- Labor relation
- Letter Recognition
- Primary Tumor => very low results
- Sonar
- Soybean
- Vehicles
- Vote
- Vowel
- Zoo

IRIS

- Normal parameters
  - Accuracy 94% with 10-fold-cross-val
- Optimizations run parameter: 100
  - Accuracy 94.67%

Colic

- Normal parameters
  - Accuracy 84.23 % with 10-fold-cross-val

- Optimizations run parameter
  - More rules discovered
  - Accuracy 85.05%, Opt. value 50
  - Accuracy 85.32%, Opt. value 100

Credit card - Germany

- Normal parameters
  - Accuracy 71.7 % with 10-fold-cross-val
- Optimizations run parameter
  - Accuracy 72.6%, Opt. value 50

Diabetes

- Normal parameters
  - Accuracy 76.04 % with 10-fold-cross-val
- Optimizations run parameter
  - No improvement is possible

Hepatitis

- Normal parameters
  - Accuracy 78.0 % with 10-fold-cross-val

- Optimizations run parameter
  - Rules are more complex and complete
  - Accuracy 80%, Opt. value 50
  - Accuracy 81.9%, Opt. value 100
  - Accuracy 82.58%, Opt. value 500

Sonar

- Normal parameters
  - Accuracy 73% with 10-fold-cross-val
- Optimizations run parameter
  - More rules are discovered
  - Rules are more complex and complete
  - Accuracy 79%, Opt. value 50

ILP

- http://www.doc.ic.ac.uk/~shm/applications.html

- Learning drug structure-activity rules:
  - General introduction to the drug structure-activity problem
  - In drugs for Alzheimer's disease
  - For drugs for inhibition of E. Coli Dihydrofolate Reductase
  - In suramin analogues

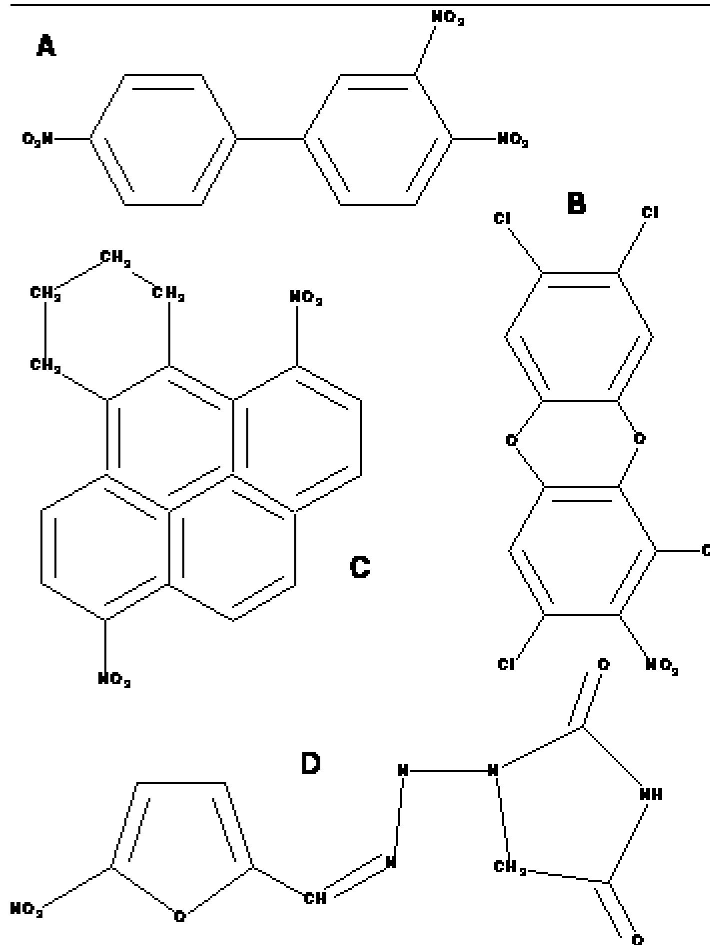
- Learning rules for predicting mutagenesis
- Learning rules for predicting protein secondary structure
- Learning rules from chess databases
- Inductive Learning of Chess Rules Using Progol

INTHELEX

- Incremental Theory Learner from Examples.
  - An ILP system developed at LACAM Lab. University of Bari, Italy.

- Experiments with Mutagenesis
  - Discretized version of dataset (Paper by (Biba, et al, ILP'2006)
  - File training: s2.tun
  - File testing: s2.tst

Mutageneis



Readings for this part

- Machine Learning. T. Mitchell
  - Chapter 10