UNIT-4

We can learn sets of rules by using **ID3 and then converting the tree to rules**. We can also use a genetic algorithm that encodes the rules as bit strings. But, these only work with predicate rules (no variables). They also consider the set of rules as a whole, not one rule at a time.

Learning Sets of Rules are

1. Sequential Covering Algorithms
2. Learning First Order Rules
3. FOIL
4. Induction as Inverted Deduction
5. Inductive Logic Programming

Sequential Covering Algorithms

Sequential Covering is a popular algorithm based on Rule-Based Classification used for learning a disjunctive set of rules. The basic idea here is to learn one rule, remove the data that it covers, then repeat the same process. In this process, In this way, it covers all the rules involved with it in a sequential manner during the training phase.

**Algorithm Involved:**

**Sequential\_covering (Target\_attribute, Attributes, Examples, Threshold):**

Learned\_rules = {}

Rule = Learn-One-Rule(Target\_attribute, Attributes, Examples)

while Performance(Rule, Examples) > Threshold :

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 according to performance over Examples

return Learned\_rules

The Sequential Learning algorithm takes care of to some extent, the low coverage problem in the Learn-One-Rule algorithm covering all the rules in a sequential manner.

**Working on the Algorithm:**

**Step 1 – create an empty decision list, ‘R’.**

**Step 2 – ‘Learn-One-Rule’ Algorithm**

**It extracts the best rule for a particular class ‘y’, where a rule is defined as: (Fig.2)**

**General Form of Rule**



**Step 2.a – if all training examples ∈ class ‘y’, then it’s classified as positive example.**

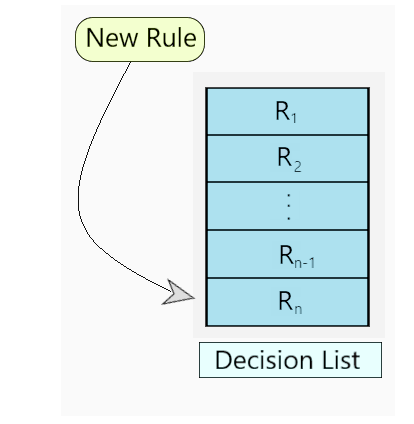
**Step 2.b – else if all training examples ∉ class ‘y’, then it’s classified as negative example.**

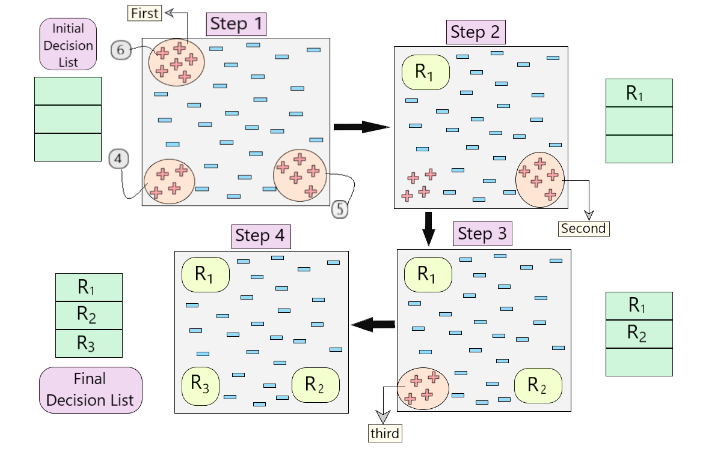
**Step 3 – The rule becomes ‘desirable’ when it covers a majority of the positive examples.**

**Step 4 – When this rule is obtained, delete all the training data associated with that rule.**

**(i.e. when the rule is applied to the dataset, it covers most of the training data, and has to be removed)**

**Step 5 – The new rule is added to the bottom of decision list, ‘R’. (Fig.3)**





* Let us understand step by step how the algorithm is working in the example shown in Fig.4.
* First, we created an empty decision list. During Step 1, we see that there are three sets of positive examples present in the dataset. So, as per the algorithm, we consider the one with maximum no of positive example. *(6, as shown in Step 1 of Fig 4)*
* Once we cover these 6 positive examples, we get our first rule R1, which is then pushed into the decision list and those positive examples are removed from the dataset. *(as shown in Step 2 of Fig 4)*
* Now, we take the next majority of positive examples *(5, as shown in Step 2 of Fig 4)*and follow the same process until we get rule R2. (Same for R3)
* In the end, we obtain our final decision list with all the desirable rules.

## Learning Rule Sets Summary

* A **sequential covering** algorithm learns one rule at a time, removing the covered examples and repeating with the rest.
* Meanwhile, **simultaneous covering** algorithms like ID3 learn the entire set of disjuncts simultaneously.
* Which is better?
* ID3 chooses attributes by comparing the partitions of the data they generate.
* CN2 chooses among alternative attribute-value pairs by comparing the subsets of data they cover.
* Thus, CN2 makes a larger number of independent choices. So it is better if there is plenty of data.
* Learn-One-Rule searches from general to specific.
* Find-S searches from specific to general.
* There are many maximally specific, but only one maximally general.
* Learn-One-Rule is generate-then-test search.
* It could be example-driven, where individual training examples constrain the generation of hypotheses.
* In generate-then-test each choice in the search is based on the hypothesis performance over many examples, so impact of noisy data is minimize. Noise can have large impact in example-driven.

Learning First Order Rules

1. Try to learn sets of rules such as

∀ x, y : Ancestor(x,y) ← Parent(x,y)  
∀ x, y : Ancestor(x,y) ← Parent(x,z) ∧ Ancestor(z,y)

1. This looks a lot like a Prolog program.

Ancestor(x,y) :- Parent(x,y)  
Ancestor(x,y) :- Parent(x,z), Ancestor(z,y)

1. Which is why inductive learning of first-order rules is often referred to as inductive logic programming.
2. Classifying web page A:

course(A) ← has-word(A, instructor) ∧¬has-word(A,good) ∧ link-from(A, B) ∧ has-word(B, problem) ∧ ¬link-from(B, C)

### First-Order Logic Definitions

1. Every expression is composed of constants, variables, predicates, and functions.
2. A term is any constant, or variable, or any function applied to any term
3. A literal is any predicate (or its negation) applied to any set of terms.

Female(Mary), ¬Female(x)

1. A ground literal is a literal that does not contain any variables.
2. A clause is any disjunction of literals whose variables are universally quantified.∀ x : Female(x) ∨ Male(x)
3. A Horn clause is an expression of the form

H ← L1 ∧ L2 ∧ ... ∧ Ln

1. For any A and B

A ← B ⇒ A ∨ ¬ B

1. so the horn clause above can be re-written as

H ∧ ¬L1 ∨ ¬L2 ∨ ... ∨ ¬Ln

1. A substitution is any function that replaces variables by terms e.g.,

{x/3, y/z} replaces x for 3 and y for z.

1. A unifying substitution θ for two literals L1 and L2 is one where

L1θ = L2θ

### Learning First-Order Horn Clauses

* Say we are trying to learn the concept Daughter(x,y) from examples.
* Each person is described by the attributes: Name, Mother, Father, Male, Female.
* Each example is a pair of instances, say a and b:
* Name(a) = Sharon, Mother(a) = Louise, Father(a) = Bob, Male(a) = False, Female(a) = True  
  Name(b) = Bob, Mother(b) = Nora, Father(b) = Victor, Male(b) = True, Female(b) = False, Daughter(a,b) = True
* If we give a bunch of these examples to CN2 or C4.5 they will output a set of rules like:
* **IF** Father(a) = Bob ∧ Name(b) = Bob ∧ Female(a) **THEN** Daughter(a,b)
* A first-order learner would output more general rules like
* **IF** Father(x) = y ∧ Female(x) **THEN** Daughter(x,y)

**FOIL**

FOIL(target-predicate, predicates, examples)  
  pos = those examples for which target-predicate is true  
  teg = those examples for which target-predicate is false  
  learnedRules = {}  
  **while** pos **do**  
    *;learn a new rule*  
    newRule = the rule that predicts target-predicate with no preconditions  
    newRuleNeg = neg  
    **while** newRuleNeg **do**  
      *;add a new literal to specialize newRule*  
      candidateLiterals = candidate new literals for newRule based on predicates  
      bestLiteral = argmax l ∈candidateLiterals Foil-gain(l, newRule)  
      add bestLiteral to preconditions of newRule  
      newRuleNeg = subset of newRuleNeg that satisfies newRuleNeg preconditions  
     learnedRules = learnedRules + newRule  
     pos = pos - {member of pos covered by newRule}  
   **return** learnedRules

* Its a natural extension of Sequential-Covering and Learn-One-Rule.
* Each iteration of the outer loop adds a new rule to its disjunctive hypothesis Learned-rules (specific-to-general).
* In the inner loop we add conjunctions that form the preconditions of the rule (general-to-specific hill-climbing).

**First-Order Logic:**

All expressions in first-order logic are composed of the following attributes:

1. constants — e.g. tyler, 23, a
2. variables — e.g. A, B, C
3. predicate symbols — e.g. male, father (True or False values only)
4. function symbols — e.g. age (can take on any constant as a value)
5. connectives — e.g. ∧, ∨, ¬, →, ←
6. quantifiers — e.g. ∀, ∃

### Generating Candidate Specializations

* How do we generate Candidate-literals?
* Suppose the current rule (NewRule) is P(x1,x2,...xk) ←L1∧...∧Ln.
* FOIL considers new literals Ln+1 of the following form
  + Q(v1,...,vr) where Q is any predicate in Predicates and vi are variables, at least one of them must already exist in the rule.
  + Equal(x,y) where x and y are variables already in the rule.
  + The negation of any of these two.

### FOIL Example

* Say we are tying to predict the Target-predicate GrandDaughter(x,y).
* FOIL begins with  
  NewRule = GrandDaughter(x,y) ←
* To specialize it, generate these candidate additions to the preconditions:  
  Equal(x,y), Female(x), Female(y), Father(x,y), Father(y.x), Father(x,z), Father(z,x), Father(y,z), Father(z,y)  
  and their negations.
* FOIL might greedily select Father(x,y) as most promising, then  
  NewRule = GrandDaughter(x,y) ← Father(y,z).
* Foil now considers all the literals from the previous step as well as:  
  Female(z), Equal(z,x), Equal(z,y), Father(z,w), Father(w,z)  
  and their negations.
* Foil might select Father(z,x), and on the next step Female(y) leading to  
  NewRule = GrandDaughter (x,y) ← Father(y,z) ∧ Father(z,x) ∧ Female(y)
* If this covers only positive examples it terminates the search for further specialization.
* FOIL now removes all positive examples covered by this new rule. If more are left then the outer loop continues.

### Search in FOIL

* Again, we are trying to learn set of rules for Target-predicate = GrandDaughter(x,y).
* Let the Examples contain  
  GrandDaughter(Victor,Sharon), Father(Sharon, Bob), Father(Tom, Bob), Female(Sharon), father(Bob, Victor)
* To select best specialization, FOIL considers ways to bind the variables. With 4 constants (Victor, Sharon, Tom, and Bob) and 2 variables (x,y) we have 4\*4 = 16 possible bindings.
* {x/Victor, y/Sharon} is the only **positive example binding** for the rule "GrandDaughter(x,y) ←". The other 15 are **negative bindings**.
* At each step, each rule is evaluated based on the sets of positive and negative variable bindings. Which one do we pick?

### Foil-Gain

* We select the literal with biggest gain Foil-Gain (L,R)≡t( log 2 p 1 p 1 +n 1 -log 2 p 0 p 0 +n 0 ) 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 is the number of positive bindings of R also covered by R +L
* It's interesting to note that - log 2 p 0 p 0 +n 0 is the optimal number of bits needed to indicate the class of a positive binding covered by R
* FOIL extends CN2 to handle first-order formulas.
* It does a general-to-specific search search, adding a single new literal to the preconditions at each step.
* The Foil-Gain function is used to select the best literal.
* It can learn recursive rules.
* If data is noisy, the search will continue until some trade-off occurs between rule accuracy, coverage, and complexity.
* FOIL stops when the length of the length of the rule is larger than the data.
* FOIL also post-prunes each rule it learns, using the same strategy as ID3 (both by Quinlan).

### Learning Recursive Rule Sets

* If we include the target predicate in Predicates then FOIL will consider it.
* This allows for the formation of recursive rules  
  Ancestor(x,y) ← Parent(x,y)  
  Ancestor(x,y) ← Parent(x,z) ∧ Ancestor(z,y)
* So, it is possible.

## Induction As Inverted Deduction

* Induction is finding h such that ∀ ⟨ x i ,f(x i)⟩∈D B∧h∧x i →f(x i ) where x i is i th training instance  
  f (x i) is the target function value for x i  
  B is other background knowledge
* Design an inductive algorithm by inverting the operators for automated deduction.

### 7.1 Inverted Example

* Target concept is Child(u,v) s.t. the child of u is v.
* We are given the single positive example ( x i )  
  Child(Bob,Sharon)  
  and the data is described by  
  Male(Bob), Female(Sharon), Father(Sharon,Bob).
* We have the general background knowledge ( B )  
  Parent(u,v) ← Father(u,v).
* So we have:  
  x i : Male(Bob), Female(Sharon), Father(Sharon,Bob)  
  f (x i) : Child(Bob,Sharon)  
  B : Parent(u,v) ← Father(u,v)
* What satisfies ∀ ⟨ x i ,f(x i)⟩∈D B∧h∧x i →f(x i ) ?  
  h 1 : Child(u,v) ← Father(v,u)  
  h 2 : Child(u,v) ← Parent(v,u)
* Notice that h 1 does not require B .
* This process of augmenting the set of predicates, based on background knowledge, is often referred to as **constructive induction**.

### Induction and Deduction

* The relationship between these two has been known for a while.

*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; ... it must be allowed that inductive investigations are of a far higher degree of difficulty and complexity than any questions of deduction...  
(Jevons 1874)*

* There are many well-known algorithms for deduction in first-order logic. Can we reverse them?

### Inverse Entailment

* An **inverse entailment operator** O(B,D) takes the training data D and background knowledge B as input and outputs a hypothesis h such that O (B,D)=h where ∀ ⟨ x i ,f(x i )⟩∈D(B∧h∧x i )→f(x i )
* Of course, there will usually be many h that satisfy this.
* A common heuristic is to use the Minimum Description Length principle.

### Inverse Entailment Pros and Cons

Pros:

* Subsumes earlier idea of finding h that “fits” training data
* Domain theory B helps define meaning of “fit” the data B ∧h∧x i →f(x i )
* Suggests algorithms that search H guided by B

Cons:

* Doesn't allow for noisy data. Consider ∀ ⟨ x i ,f(x i )⟩∈D (B∧h∧x i )→f(x i )
* First order logic gives a huge hypothesis space H . This leads to over-fitting and the intractability of calculating all acceptable h 's.
* The complexity of space search increases as the background knowledge B is increased.

### Resolution Rule

* The resolution rule is a [**sound**](http://www.wikipedia.org/wiki/Soundness) [6] and [**complete**](http://www.wikipedia.org/wiki/Completeness) [7] rule for deductive inference in first-order logic.
* The rule is:  
    P ∨ L  
  ¬L ∨ R  
  --------  
  P ∨ R
* More generally
  1. Given initial clauses C1 and C2, find a literal L from C1 such that ¬L is in C2.
  2. Form the resolvent C by including all literals from C1 and C2 except for L and ¬L:  
     C = (C1 - {L}) ∪ (C2 - {¬L})

### Inverting Resolution

* The inverse entailment operator must derive C2 given the resolvent C and C1.
* Say C = A ∨ B, and C1= B ∨ D. How do we derive C2 s.t. C1 ∧ C2 → C?
* Find the L that appears in C1 and not in C, then form C2 by including the following literals  
  C2 = (C - (C1 - {L})) ∪ {¬ L}  
  so  
  C2 = A ∨ ¬D
* C2 can also be A ∨ ¬D ∨ B.
* In general, inverse resolution can produce multiple clauses C2.

### Learning With Inverted Resolution

* Use inverse entailment to construct hypotheses that, together with the background information, entail the training data.
* Use sequential covering algorithm to iteratively learn a set of Horn clauses in this way.

1. Select a training example that is not yet covered by learned clauses.
2. Use inverse resolution rule to generate candidate hypothesis h that satisfies B ∧ h ∧ x → f(x), where B = background knowledge plus any learned clauses.

* This is example-driven search.
* If multiple candidate hypotheses then choose one with highest accuracy over the other examples.

### First-Order Resolution

* In general
  1. Find a literal L1 from clause C1, L2 from clause C2, and substitution θ such that L1 θ = ¬ L2 θ.
  2. Form the resolvent C by including all literals from C1θ and C2θ, except for L1θ and ¬L2θ. More precisely, the set of literals occurring in the conclusion C is  
     C = (C1 - {L1})θ ∪(C2 - {L2})θ
* That is, θ is a unifying substitution for L1 and ¬ L2.
* For example, let C1 = White(x) ← Swan(x) and C2 = Swan(Fred).  
  C1 can be re-written as White(x) ∨ ¬ Swan(x)  
  Then L1 = ¬ Swan(x), L2 = Swan(Fred) if θ ={x/Fred}  
  So C = White(Fred).

### Inverting First-Order Resolution

* θ can be uniquely factored into θ1 and θ2 where θ1 contains all the substitutions involving variables from C1, and θ2 for C2.
* So now,  
  C = (C1 - {L1})θ1 ∪ (C2 - {L2})θ2  
  which can be re-written as  
  C - (C1 - {L1})θ1 = (C2 - {L2})θ2  
  then by definition we have that L2 = ¬L1 θ1 θ2-1 so  
  C2 = (C - (C1 - {L1}θ1)θ2-1 ∪ {¬L1 θ1 θ2-1}
* In applying this we will find multiple choices for L1, θ1, and θ2.

### Inverted First-Order Example

* Target concept = GrandChild(x,y)  
  D = GrandChild(Bob, Shanon)  
  B = {Father(Shanon,Tom), Father(Tom,Bob)}.
* C = GrandChild(Bob,Shanon)  
  C1 = Father(Shanon, Tom)  
  L1 = Father(Shanon, Tom)  
  θ1-1 = {}  
  θ2-1 = {Shanon/x}
* Then, the resulting C2 is  
  GrandChild(Bob,x) ∨ ¬Father(x,Tom)
* This inferred clause may now be used as the conclusion C for a second inverse resolution.

### Inverse Resolution Summary

* Generate hypothesis h that satisfy the constraint B ∧h∧x i→f(x i)
* Many might be generated, but they all satisfy the equation, unlike FOIL.
* Still, search is often unfocused and inefficient because it only considers a small fraction of the data when generating the hypothesis.

## Generalization, θ-Subsumption, and Entailment

* **more-general-than**: Given two boolean functions h i (x) and h j (x) we say that h i is more general than h j if ∀ x:h j(x)→h i(x) (used by Candidate-Elimination).
* **θ-subsumption**: Clause C1 θ-subsumes C2 iff there exists a θ such that C1θ ⊆ C2 (i.e., the set of literals is a subset).
* **Entailment** C1 entails C2 iff C2 follows deductively from C1.
* more-general-than is a special case of θ-subsumption which is a special case of entailment.

## PROGOL

* The idea in the PROGOL system is to reduce the combinatorial explosion by generating the most specific acceptable h .
* User specifies H by stating predicates, functions, and forms of arguments allowed for each.
* PROGOL uses sequential covering algorithm. For each ⟨ x i,f(x i)⟩ it finds the most specific hypothesis h i s.t. B ∧h i∧x i →f(x i ) . (actually, it considers only k -step entailment).
* It then conducts a general-to-specific search bounded by specific hypothesis h i , choosing hypotheses with minimum description length.