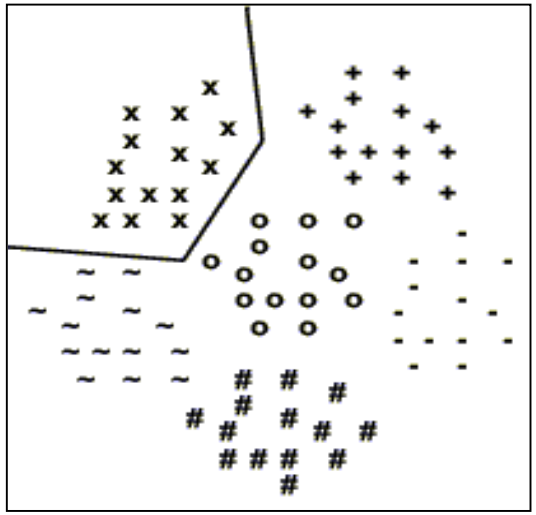
Inductive Rule Learning



Univ.-Lektor Dr.techn. Alexander K. Seewald
Österreichisches Forschungsinstitut
für Artificial Intelligence

alex@seewald.at / alex.seewald.at

Revisited: How to improve on 1R?

How to reduce 1R's bias / increase its complexity?

- **Divide-and-Conquer:** Apply algorithm recursively! Choose best attribute at top and then recursively create rules for each subset instead of just counting the most common class. Repeat until "pure" (=only examples of the same class). This creates a *decision tree* of attribute and class values.

⇒ **Decision Tree Learning** (as we already saw)

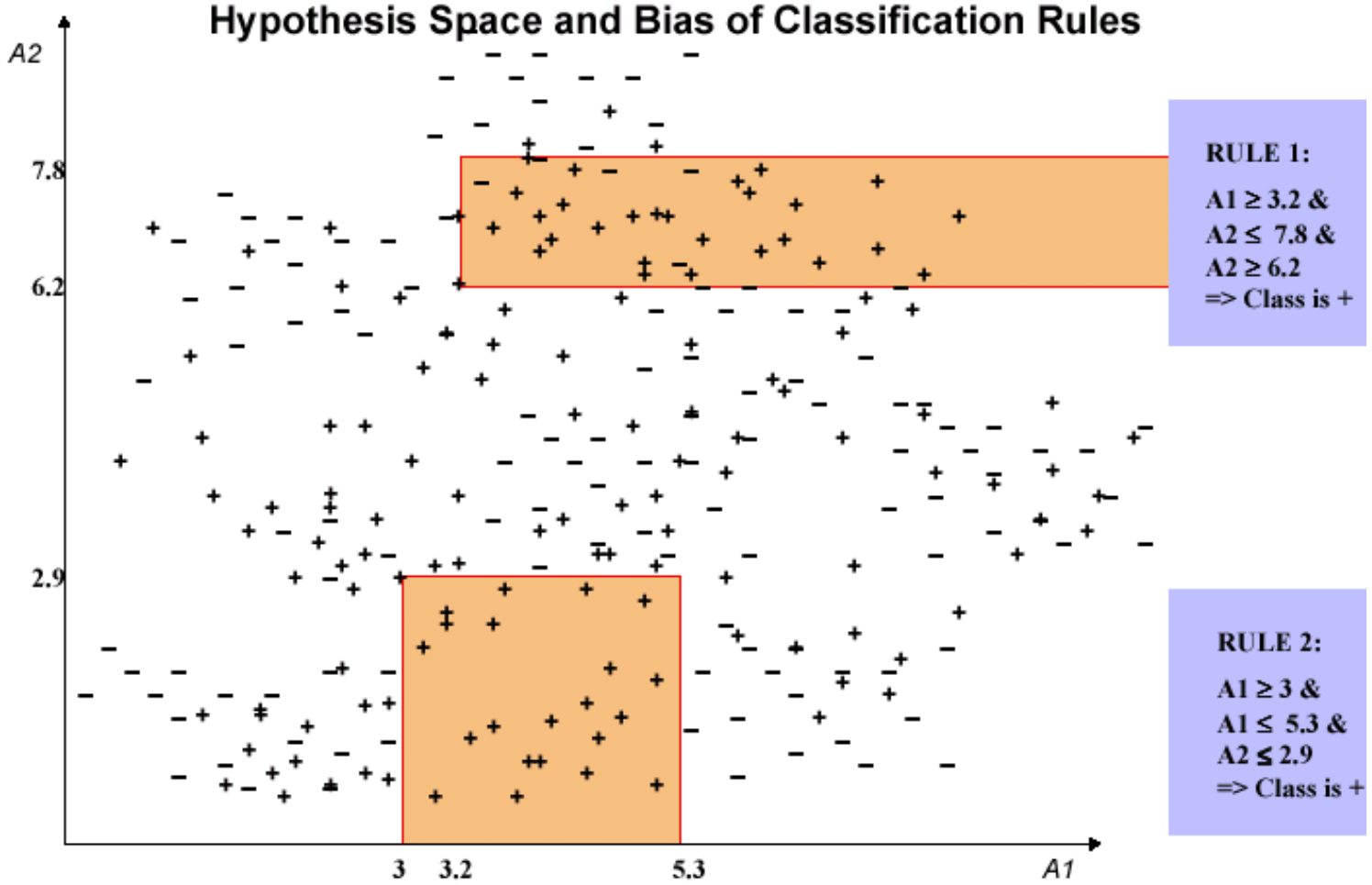
Another idea:

- **Separate-and-Conquer:** Learn *best* rule for a subset of training data, remove both *positive* examples which are correctly classified and those negative examples erroneously classified as positive (latter examples should number few or none). Apply this algorithm recursively until very few or no positive examples remain. This creates a *rule set* for classification.
- Not as efficient as divide-and-conquer, but has other advantages: More compact than decision trees, a modular and easily understandable representation and the ability to learn partial models.

⇒ **Inductive Rule Learning**

alex@seewald.at / alex.seewald.at

Rule Sets - Bias & Variance



Low bias, high variance. Concept boundaries are axis-parallel hyperrectangles (see above), one for each rule within the ruleset.

alex@seewald.at / alex.seewald.at

Separate-And-Conquer Rule Learning

Learning Rule Sets: Sequential Set Covering Algorithm

```
procedure LearnRules(c,TD)
Rules := {}; Pos := {x_i ∈ TD | y_i = c};
Neg := {x_i ∈ TD | y_i ≠ c};
repeat
    Rule := FindBestRule(c,Pos,Neg);
    Pos := Pos \ Covers(Rule,Pos);
    Neg := Neg \ Covers(Rule, Neg);
    Rules := Rules ∪ Rule;
until RuleSetStopCrit(Rules,Pos,Neg);
return (Rules);
```

Simplest stopping criterion 1:

RuleSetStopCrit = true if all positive examples are covered by the current rule set, i.e. *Pos* = {}

```
procedure FindBestRule(c,Pos,Neg)
Rules := { ⇒ c }    // ← create default rule
while not RuleStopCrit(Rule,Pos,Neg) do
    Rule := Rule ∪ FindBestCondition(Pos,Neg);
    Pos := Covers(Rule,Pos);
    Neg := Covers(Rule,Neg);
endwhile; return (Rule);
```

Simplest stopping criterion 2:

RuleStopCrit = true if the rule covers no more negative examples, i.e., *Neg* = {}. Such a rule is called **consistent**.

A rule **covers** an example if the conditions of the rule match the attribute values of the example. Rules always predict the *positive* class.

alex@seewald.at / alex.seewald.at

How to find the best rule?

Given set of positive / negative examples, find best rule (=set of conditions) that

- Covers many positive examples
- Covers few or none negative examples
- Is as simple as possible (Ockham's razor)

Exhaustive search is impossible since this would have to consider all possible subsets of rule conditions (n^p for p nominal attributes each having n-1 possible values). Heuristic search is needed, but gives no guarantee of best solution.

General search directions

- **Bottom-up** (specific-to-general)

  Start with a very long list of conditions (e.g. the complete description of one positive example) and delete conditions one by one (step-wise generalization)
- **Top-down** (general-to-specific)

  Start with empty rule (no conditions) and add conditions one by one (step-wise specialization). Similar to our version of *FindBestRule*.

alex@seewald.at / alex.seewald.at

Heuristic Search Algorithms

- **Hill-climbing**

  At each step, add or drop the condition (from all possible conditions) that maximises some local heuristic evaluation measure.

  Problem: short-sighted and greedy.

- **Beam search**

  Always keep a list of n alternative refinements; expand the currently best one (according to some local heuristic). Explores larger portion of search space and can find globally better solutions. Less short-sighted, but still greedy.

- **Best-first search**

  Explore all possible solutions, always focusing on the most promising first. If unrestricted, explores full search space. Must be accompanied by search pruning. Still inefficient, but can guarantee to find best solution.

...

Most commonly used

- Top-down search with hill-climbing (as in *FindBestRule*) or beam search.

alex@seewald.at / alex.seewald.at

How to find best condition to add?

General approach for *FindBestCondition*: test all possible conditions and choose the *best* one according to heuristic, e.g. max.ent. & (p>n).

Example: Weather dataset, *yes*=positive examples (target), *no*=negative examples.

- Partial rule {outlook=rainy ⇒ *yes*}. Covers 5 examples: **3 positive** and **2 negative**.

Possible conditions for improvements:

Temp < 66.5 (0+,1-)	Humidity ≥ 75 (3+,1-)
Temp ≥ 66.5 (3+,1-)	Humidity < 75 (0+,1-)
Temp < 70.5 (2+,1-)	Windy=false (3+,0-)
Temp ≥ 70.5 (1+,1-)	Windy=true (0+,2-)
Temp < 73.0 (2+,2-)	...
Temp ≥ 73.0 (1+,0-)	

- Choose Windy=false. Refined rule is now {outlook=rainy & windy=false ⇒ *yes*}. Covers 3 positive and 0 negative examples.

⇒ **We have found a consistent rule and return it**

Outlook	T	H	Windy	Play?
overcast	64°F	65%	true	*yes*
rainy	**65°F**	**70%**	**true**	***no***
sunny	69°F	70%	false	*yes*
sunny	75°F	70%	true	*yes*
overcast	81°F	75%	false	*yes*
rainy	**68°F**	**80%**	**false**	***yes***
rainy	**75°F**	**80%**	**false**	***yes***
sunny	85°F	85%	false	*no*
overcast	83°F	86%	false	*yes*
sunny	80°F	90%	true	*no*
overcast	72°F	90%	true	*yes*
rainy	**71°F**	**91%**	**true**	***no***
sunny	72°F	95%	false	*no*
rainy	**70°F**	**96%**	**false**	***yes***

alex@seewald.at / alex.seewald.at

Heuristic Evaluation Functions

Notation:

P ... the total number of positive examples in training data TD

N ...the total number of negative examples in training data TD

r' ... the current (incomplete) rule

p' ... the number of positive examples covered by *r'*

n' ... the number of negative examples covered by *r'*

r ... the rule resulting from adding a condition to *r'*

p ... the number of positive examples covered by *r*

n ... the number of negative examples covered by *r*

alex@seewald.at / alex.seewald.at

Heuristic Evaluation Functions

Basic Heuristics

Positive Coverage: $C(r) = p$

Accuracy: $A(r) = \frac{p + (N - n)}{P + N} \cong p - n$

Purity: $P(r) = \frac{p}{p + n}$

Information Content: $IC(r) = -\log_2 P(r)$

Entropy: $E(r) = -\frac{p}{p+n}\log_2 \frac{p}{p+n} - \frac{n}{p+n}\log_2 \frac{n}{p+n}$

Cross Entropy: $CE(r) = -\frac{p}{p+n}\log_2 \frac{\frac{p}{p+n}}{\frac{P}{P+N}} - \frac{n}{p+n}\log_2 \frac{\frac{n}{p+n}}{\frac{N}{P+N}}$

alex@seewald.at / alex.seewald.at

Heuristic Evaluation Functions

Basic Heuristics (2)

Laplace Estimate : $$LAP(r) = \frac{p+1}{p+n+2}$$

$m-$ Estimate : $$M(r) = \frac{p + m\frac{P}{P+N}}{p+n+m}$$

Correlation : $$Corr(r) = \frac{\frac{p+tn-fn-n}{p'+n'} - \left(\frac{p'-n'}{p'+n'}\right)\left(\frac{p+n-(tn+fn)}{p'+n'}\right)}{\left(1-\left(\frac{p'-n'}{p'+n'}\right)^2\right)\left(1-\left(\frac{p+n-(tn+fn)}{p'+n'}\right)^2\right)}$$

$$tn = n'-n \quad fn = p'-p$$

Gain Heuristics

Coverage Gain : $$CG(r) = \frac{p-p'}{P} - \frac{n-n'}{N}$$

Weighted Information Gain : $$WIG(r) = -C(r)(IC(r) - IC(r'))$$

alex@seewald.at / alex.seewald.at

Overfitting Avoidance: Pre-Pruning

Basic ideas

- Stop refining a rule although it is still inconsistent (i.e. covers neg. instances)
- Stop adding new rules although some positive instances are still not covered.

Basic method

- Modify stopping criteria *RuleSetStopCrit*, *RuleStopCrit* in LearnRules et al.

Commonly used criteria

- **Minimum Purity:** If the best next rule that can be found is below a specified purity threshold ($Purity(r) < \varepsilon$), stop adding rules ($\Rightarrow$ RuleSetStopCrit)
- **Encoding Length Restriction:** Number of bits needed to encode a rule must be less than number of bits needed to code the covered examples. I.e., stop refining a rule when it would become too complex ($\Rightarrow$ RuleStopCrit)
- **Significance Test:** Stop adding conditions to a rule if none of the conditions shows a pre-specified minimum correlation with the class labels, similar to pre-pruning via X^2 in DT slides ($\Rightarrow$ RuleStopCrit)

alex@seewald.at / alex.seewald.at

Overfitting Avoidance: Post-Pruning

Basic idea

First learn a (possibly large) set of (possibly complex) rules that fit the training data well; then gradually simplify the rule set by

- Dropping conditions in rules ($\Rightarrow$ simplifying/generalizing the rules)
- Dropping entire rules ($\Rightarrow$ simplifying/generalizing the model)

A standard method: Reduced Error Pruning (~ DTs)

- Split training set into a *growing set* (e.g., 70%) and a *pruning set*
- Learn theory from growing set
- Simplify theory stepwise

  Consider dropping conditions and dropping rules. Always perform (greedily) the simplification step that produces the greatest improvement in e.g. accuracy on the pruning set until no step improves the rule set anymore.

alex@seewald.at / alex.seewald.at

Reduced Error Pruning for Rule Learning

Reduced Error Pruning

```
procedure REP(TD, SplitRatio)
SplitExamples( SplitRatio, TD, GrowingSet, PruningSet);
Model := LearnRules(GrowingSet);
NewModel := BestSimplification(Model, PruningSet);
while Accuracy(NewModel, PruningSet) ≥ Accuracy(Model, PruningSet)
    Model := NewModel;
    NewModel := BestSimplification(Model, PruningSet);
endwhile;
return (Model);
```

Further improvements on REP (which has a worst-case complexity of $O(n^4)$)

- Incremental REP (IREP): Prune each rule separately, removing covered examples from *GrowingSet* **and** *PruningSet*. Remaining instances are redistributed into new *Growing/PruningSet*. Stop when predictive accuracy on *PruningSet* is below baseline accuracy (i.e. accuracy of the empty rule *ZeroR*)
- Repeated Incremental Pruning to Produce Error Reduction (RIPPER): Optimized version of IREP which runs the learning process multiple times.

alex@seewald.at / alex.seewald.at

Multi-Class Learning Tasks

Two-class Rule Learning is not symmetric under class permutation!

- E.g. for the weather dataset, learning a rule set for *yes*=positive vs. *no*=negative examples is a different learning task from *yes*=negative vs. *no*=positive examples! In one case, we learn a rule set predicting Play=*yes*, in the other case we learn a rule set predicting Play=*no*. These will certainly differ, depending on how complex these two sets are.
- Most other learners (Linear Methods, Instance-Based Learning, Bayesian Methods and Decision Tree Learning) are symmetric under class permutation.

This behaviour of Rule Learning is clearly undesirable. We would like to get the *same* ruleset in both cases. The simplest way to achieve this in the two-class case is to **combine** both rule sets. This is equivalent to One-Against-All as we will see shortly.

For learning tasks with more than two classes, there are several ways to map the learning tasks to multiple binary representations.

alex@seewald.at / alex.seewald.at

Unordered vs. Ordered Rule Sets

Unordered Rule Set

- Rules can be applied in any order.
- Disjunction of conjuctive conditions to belong to a class = Disjunctive Normal Form (DNF). In real life, rules may overlap and conflict!

(Outlook = overcast) ⇒ Play? = yes
(Outlook = sunny) and (Humidity < 75) ⇒ Play? = yes
(Outlook = rain) and (Windy = false) ⇒ Play? = yes
(Outlook = sunny) and (Humidity ≥ 75) ⇒ Play? = no
(Outlook = rain) and (Windy = true) ⇒ Play? = no

Ordered Rule Set (=ordered decision list)

- Rules **must** be applied in the given order.
- First rule matching an example predicts the classification.
- Cascade of *If-Then-Else* Statements

Default rule

(Outlook = sunny) and (Humidity ≥ 75) ⇒ Play? =no
(Outlook = rain) and (Windy = true) ⇒ Play? = no
⇒ Play? = yes

alex@seewald.at / alex.seewald.at

Unordered Rule Sets: One Against All

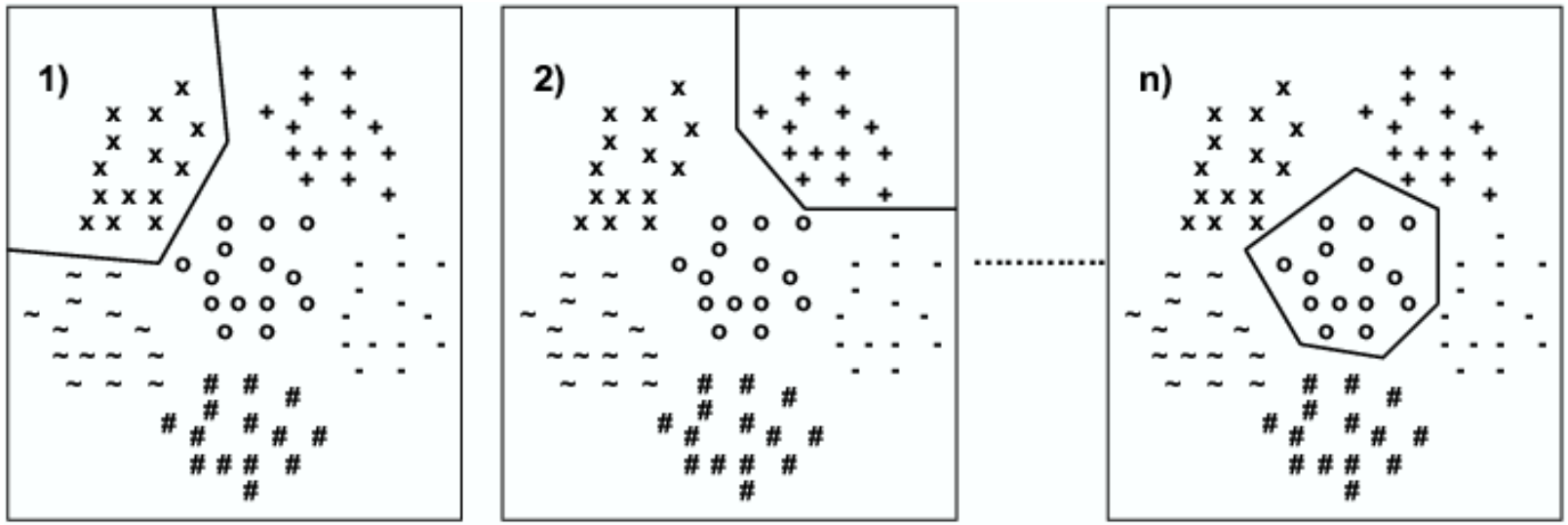
A strategy for learning an unordered rule set in multiclass learning tasks: Learn rules for one class at a time, against all others.

```
procedure LearnOneVsAll(TD, Classes)
Model := {};
foreach c in Classes
        Pos := {x_i ∈ TD | y_i = c};
        Neg := {x_i ∈ TD | y_i ≠ c};
        Rules := LearnRules(c,Pos,Neg);
        Model := Model ∪ Rules;
endfor;
return(Model);
```

Most common approach to map multiclass to binary learning tasks.
Also applicable to non-rule learners (e.g. Linear and Logistic Regression)

alex@seewald.at / alex.seewald.at

Unordered Rule Sets: One Against All



1) ..... and .... => Class = x
.... => Class = x
.....

2) ..... and .... => Class = +
.... => Class = +
.....

⋮

n) ..... and .... => Class = o
.... => Class = o
.....

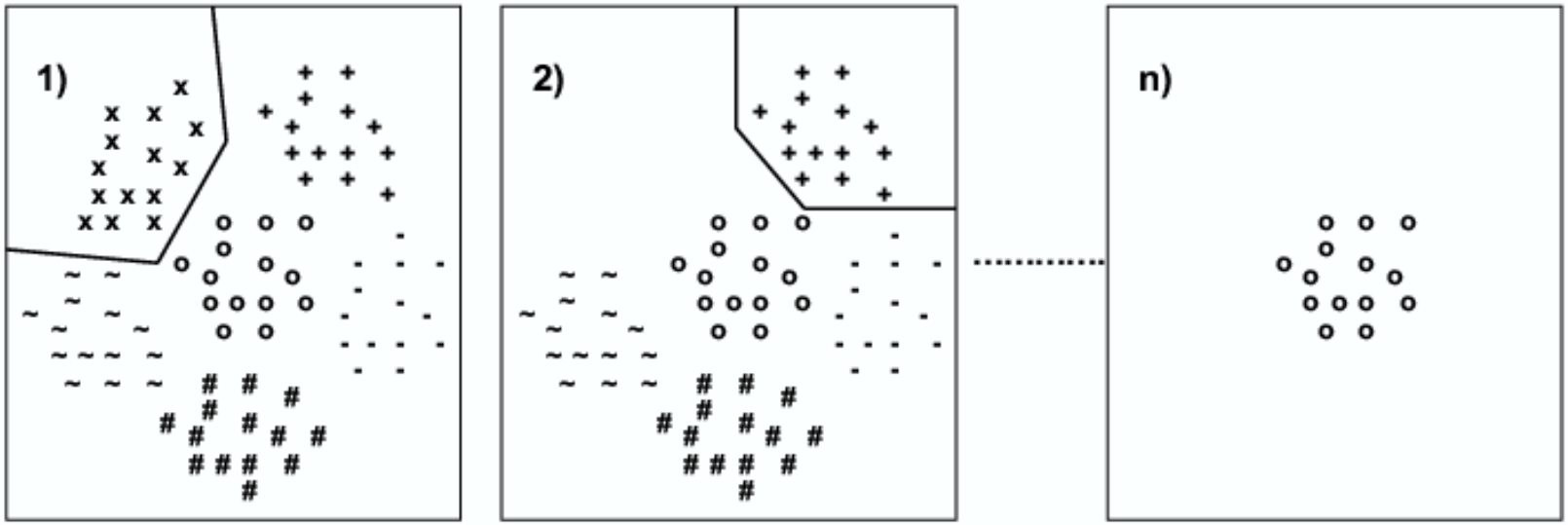
alex@seewald.at / alex.seewald.at

Ordered Rule Sets: One Against Rest

A strategy for learning an ordered rule set in multiclass learning tasks: Learn rules for one class at a time, against the remaining ones.

```
procedure LearnOneVsRest(TD, Classes)
Model := {}; RemainingClasses := Classes; Examples=TD;
foreach c in Classes
    if |RemainingClasses| =1
        then Rules := { => c }    // ← create default rule
    else
        Pos := {x_i ∈ Examples | y_i = c};
        Neg := {x_i ∈ Examples | y_i ≠ c};
        Rules := LearnRules(c,Pos,Neg);
    endif
    Model := Model ∪ Rules;
    Examples := Examples \ {x_i ∈ Examples | y_i = c};
    RemainingClasses := RemainingClasses \ {c};
endfor; return(Model);
```

Ordered Rule Sets: One Against Rest



1) ..... and .... => Class = x
.... => Class = x
.....

2) ..... and ..... => Class = +
.... => Class = +
.....

⋮

n) => Class = o (Default rule)

=> must be executed in order !

alex@seewald.at / alex.seewald.at

Round-Robin Rule Learning

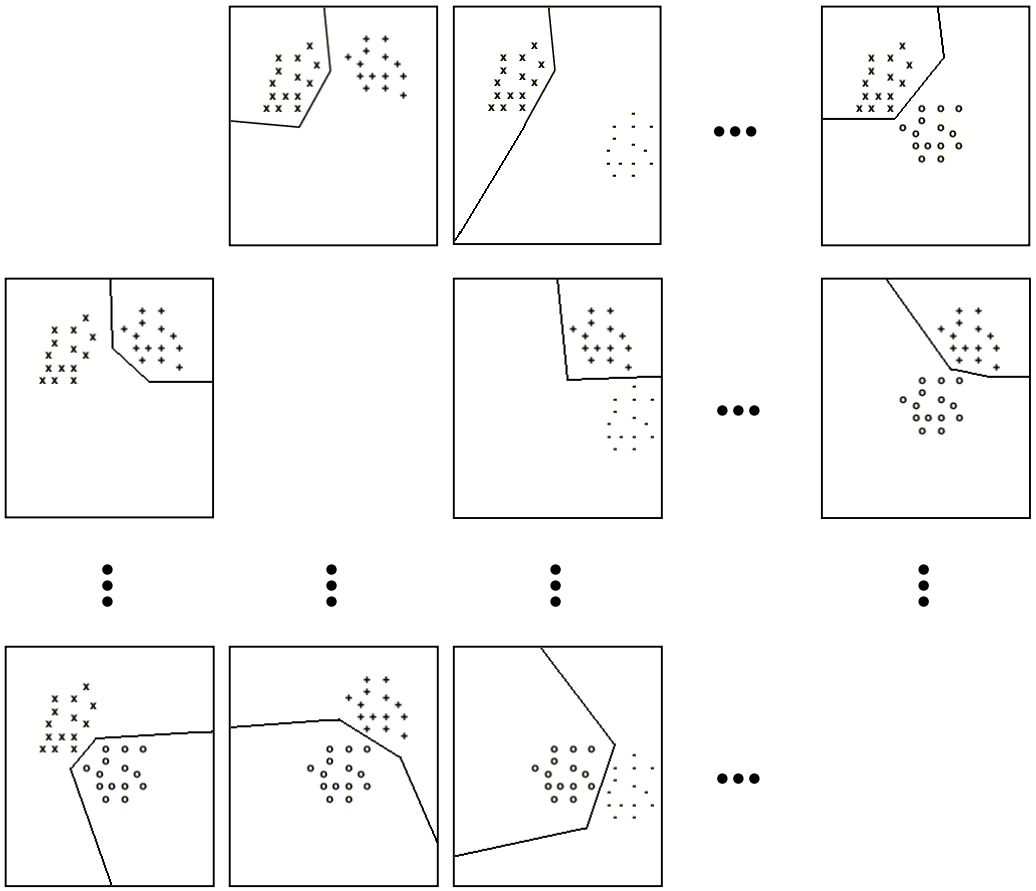
A strategy to learn Ordered and Unordered Rule Sets in multiclass learning tasks: Round Robin Rule Learning

```
procedure LearnRoundRobin(TD, Classes)
Model := {};
foreach c in Classes
    foreach d in Classes \ {c}
        Pos := {x_i ∈ TD | y_i = c};
        Neg := {x_i ∈ TD | y_i = d};
        Rules := LearnRules(c,Pos,Neg);
        Model := Model ∪ Rules;
    end;
endfor; return(Model);
```

Another approach to map multiclass to binary learning tasks. Applicable to non-rule learners (e.g. used internally in most SVM algorithms)

alex@seewald.at / alex.seewald.at

Round-Robin Rule Learning



Learn rules to discriminate each combination of classes, ignoring examples from all other classes.

For learning *unordered rulesets*, learn all combinations as shown.

For learning *ordered rulesets*, learn only combinations above the diagonal (i.e. learn only classes c_i vs c_j where i<j)

alex@seewald.at / alex.seewald.at