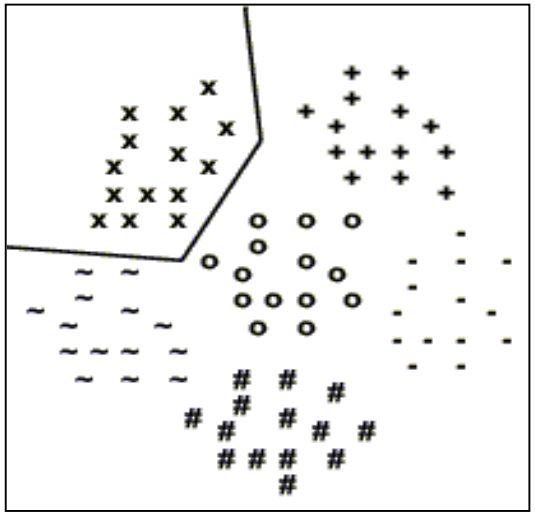
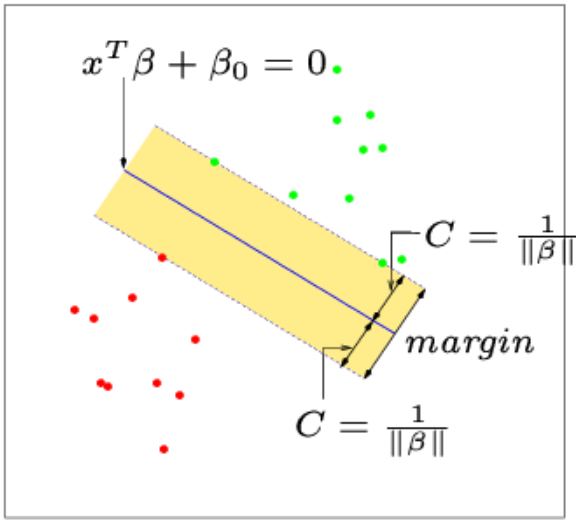
Inductive Rule Learning & Support Vector Machines





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

alex@seewald.at / alex.seewald.at

Two Ways To Learn

Divide-and-Conquer: Apply 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)

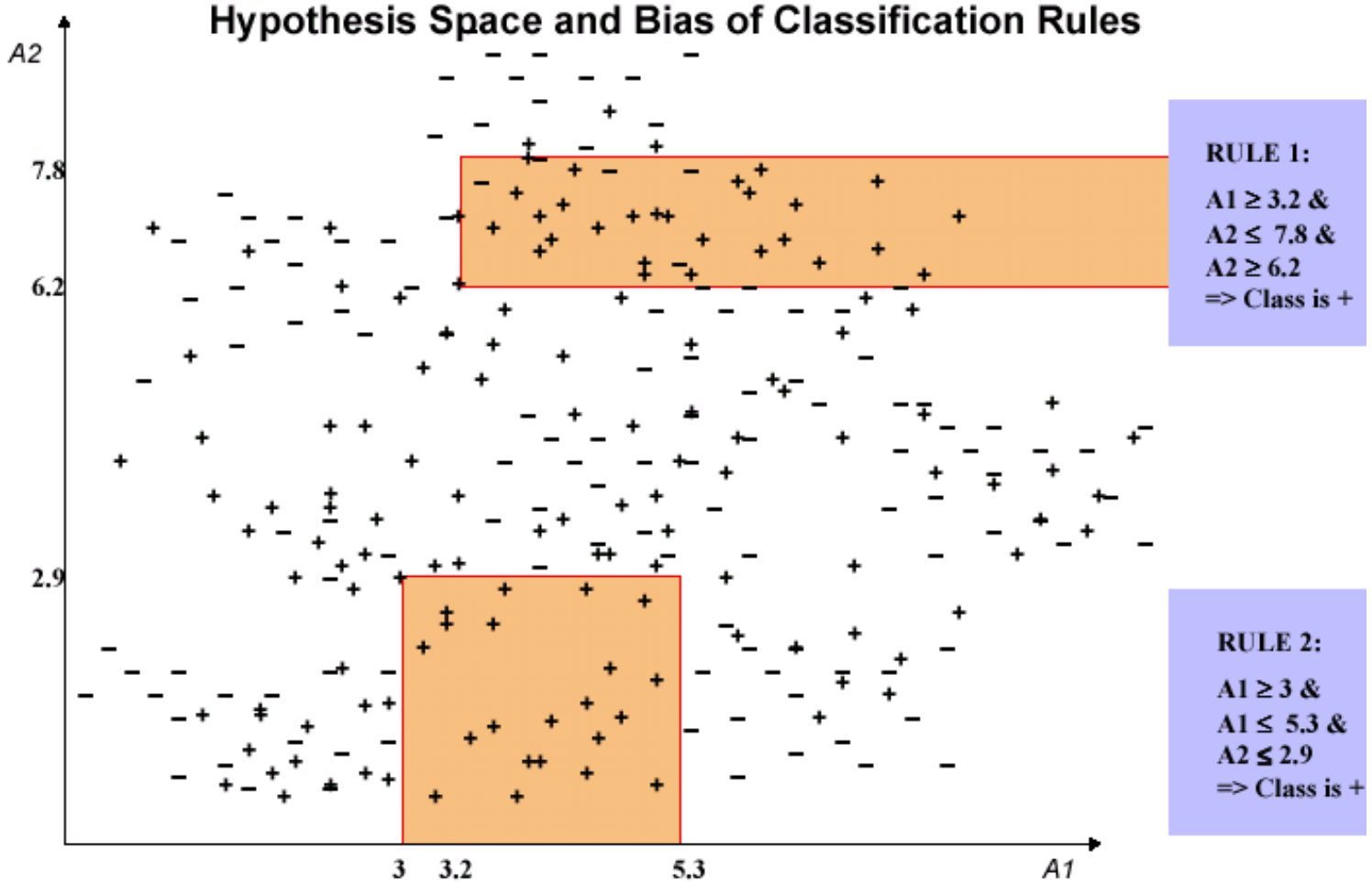
Separate-and-Conquer: Learn *best* rule for a subset of training data, remove both *positive* examples which are correctly classified and those negative examples erroneously classified as positive (latter examples should number few or none). Apply this algorithm recursively until very few or no positive 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

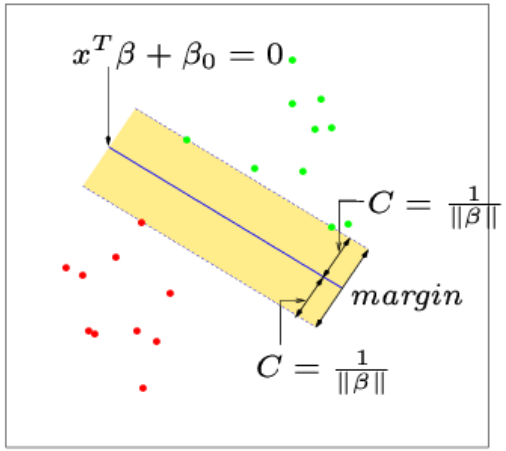
```
procedure REP(TD, SplitRatio)
SplitExamples( SplitRatio, TD, GrowingSet, PruningSet);
Model := LearnRules(GrowingSet);
NewModel := BestSimplification(Model, PruningSet);
while Accuracy(NewModel, PruningSet) ≥ Accuracy(Model, PruningSet)
    Model := NewModel;
    NewModel := BestSimplification(Model,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

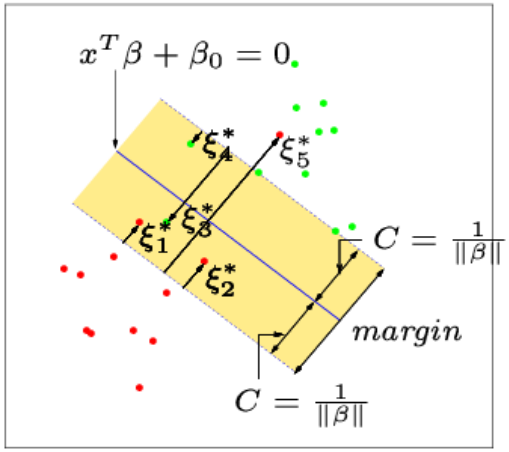
Further improvements on REP (which has a worst-case complexity of $O(n^4)$)

- 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

Support Vector Machines





Initially, a linear model in **x**. We don' t minimize residual squared error (RSS) or log-likelihood, but maximize the margin $\|\beta\|$ resp. minimize $C=1/\|\beta\|$. $f(\mathbf{x}) = \mathbf{x}^T.\beta+\beta_0$ is our function/model, where $y_i \in \{+1,-1\}$ (i.e. the sign of $f(\mathbf{x})$ determines the class)**,** but the weights β and constant term β_0 are determined differently. This new regularization again guarantees an unique solution.

- If the data is linearly separable, we minimize $\|\beta\|$ subject to the constraints $y_i(\mathbf{x_i}^T.\beta+\beta_0)\geq 1$ for $\forall$ i=1,2,...|TD|. See top left figure.
- If the data is not linearly separable, we introduce slack variables ξ_i to let some examples be on the wrong side of the margin. See top right figure.

$$\min\|\beta\| \text{ subject to } \begin{cases} y_i(\mathbf{x_i}^T\beta+\beta_0) \geq 1-\xi_i \\ \xi_i \geq 0, \sum_{i=1}^{|TD|}\xi_i \leq \text{constant} \approx \frac{1}{\lambda} \end{cases}$$

alex@seewald.at / alex.seewald.at

Support Vector Machines (2)

This optimization problem is quadratic with linear inequality constraints and is thus convex. A quadratic programming solution using Lagrange multipliers is therefore feasible. An equivalent form of the nonseparable case is:

$$\min \frac{1}{2}\|\beta\|^2 + \lambda \sum_{i=1}^{|TD|} \xi_i \text{ subject to } \xi_i \geq 0,\ y_i(\mathbf{x_i}^{\mathbf{T}}\beta + \beta_0) \geq 1 - \xi_i \ \forall i$$

Parameter λ determines the weight given to optimizing the slack variables ξ_i versus optimizing the margin. The separable case corresponds to $\lambda=\infty$.

The Lagrange primal function combines minimalization and constraints into a single formula. The constraints are weighted by Lagrange multipliers α_i and μ_i.

$$L_p = \frac{1}{2}\|\beta\|^2 + \lambda \sum_{i=1}^{|TD|} \xi_i - \sum_{i=1}^{|TD|} \alpha_i [y_i(\mathbf{x_i}^{\mathbf{T}}\beta + \beta_0) - (1 - \xi_i)] - \sum_{i=1}^{|TD|} \mu_i \xi_i$$

which will be maximized w.r.t. β, β_0 and ξ_i. Setting the derivates to zero yields:

$$\beta = \sum_{i=1}^{|TD|} \alpha_i y_i \mathbf{x_i}$$

$$0 = \sum_{i=1}^{|TD|} \alpha_i y_i$$

$$\mu_i = \lambda - \alpha_i$$

$$\alpha_i, \mu_i, \xi_i \geq 0$$

Substituting into L_P yields the Lagrange (Wolfe) dual objective function

$$L_D = \sum_{i=1}^{|TD|} \alpha_i - \frac{1}{2} \sum_{i=1}^{|TD|} \sum_{j=1}^{|TD|} \alpha_i \alpha_j y_i y_j \mathbf{x_i}^{\mathbf{T}} \mathbf{x_j}$$

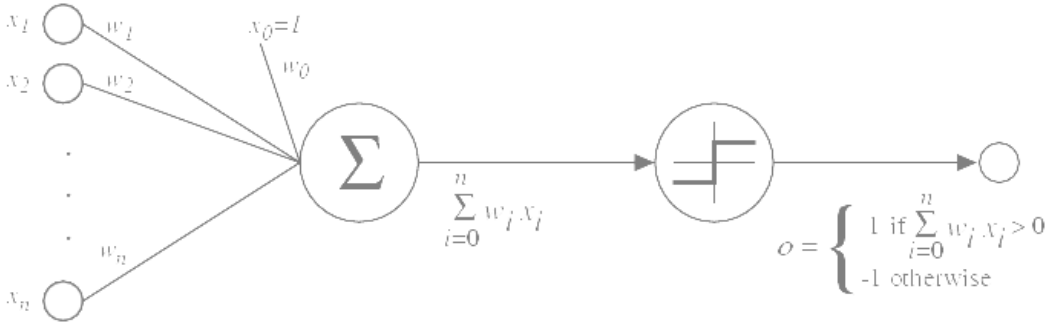
and the Karush - Kuhn - Tucker conditions (indicates a solution)

$$(\lambda - \alpha_i)\xi_i = 0;\ \alpha_i[y_i(\mathbf{x_i}^{\mathbf{T}}\beta + \beta_0) - (1 - \xi_i)] = 0$$

$$y_i(\mathbf{x_i}^{\mathbf{T}}\beta + \beta_0) - (1 - \xi_i) \geq 0 \text{ (resp. for } \forall i)$$

alex@seewald.at / alex.seewald.at

A high-bias learner: Perceptron



If the weight vector **w** is initially set to all zeros, the final **w** after convergence will be a linear combination of the training examples, similar to a SVM.

Perceptron (linear binary threshold unit, linear model)

- Computes a linear function of **x** (assume adding an $x_0=1$ to **x**, so that constant term w_0 can be handled). $f(\mathbf{x}) = \text{sign}(\mathbf{x}^T.\mathbf{w})$, **w** is initialized randomly.
- *Perceptron training rule*: $\mathbf{w} \leftarrow \mathbf{w}+\eta(y-f(\mathbf{x})).\mathbf{x}^T$, where y is the true output value from training data (±1), and η is the learning rate.
- Intuitively, concept boundary is a hyperplane which separates classes +1 & -1.

  Update rule is applied to each training example in turn, repeating until all training examples are classified correctly. Provided η is small enough, and the training set is linearly separable, this algorithm converges in a finite number of steps. If data is not linearly separable, convergence is not assured.

alex@seewald.at / alex.seewald.at

Support Vector Machines (3)

Features of the optimization problem for SVMs

- No local minima, only one global minimum – the maximal margin hyperplane.
- To solve the dual problem we need only the dot product $\mathbf{x_i}^T\mathbf{x_j}$ for each combination of training instances. The function computing the dot product, *K(u,v)* is called *kernel*. This **kernel trick** enables us to expand the original feature space via $\phi(x)$, thus learning a maximum margin hyperplane in higher-dimensional feature space which gives a nonlinear decision boundary in the original, lower-dimensional feature space. Usually only a few α_i are nonzero - the associated examples $\mathbf{x_i}$ are called *support vectors*.

$$L_D = \sum_{i=1}^{|TD|}\alpha_i - \frac{1}{2}\sum_{i=1}^{|TD|}\sum_{j=1}^{|TD|}\alpha_i\alpha_j y_i y_j K(\mathbf{x_i},\mathbf{x_j}) \text{ where } K(u,v) = \langle\phi(u)\cdot\phi(v)\rangle$$

$$\phi : \Re^p \mapsto \Re^m \ (m >> p) \text{ and } \langle\phi(u)\cdot\phi(v)\rangle = \sum_{i=1}^{m}\phi_i(u)\phi_i(v) \ = \text{the dot product.}$$

- The weight vector β is a linear combinations of training examples $\mathbf{x_i}$. We can use this relation to compute model f(**x**) via the kernel function. This allows us even an infinite-dimensional $\phi(x)$, i.e. $m=\infty$, without explicit computation of β.

$$f(\mathbf{x}) = \sum_{i=1}^{|TD|}\alpha_i y_i K(\mathbf{x},\mathbf{x_i}) + \beta_0$$

alex@seewald.at / alex.seewald.at

Example: A Polynomial Kernel of Degree 2

$$K(u,v) = \langle u \cdot v \rangle^2 = \left(\sum_{i=1}^{p} u_i v_i \right)^2 = \left(\sum_{i=1}^{p} u_i v_i \right) \left(\sum_{j=1}^{p} u_j v_j \right) \sum_{i=1}^{p} \sum_{j=1}^{p} u_i u_j v_i v_j = \sum_{(i,j)=(1,1)}^{(p,p)} (u_i u_j)(v_i v_j)$$

which is equivalent to a dot product using $\phi(u) = (u_i u_j)_{(i,j)=(1,1)}^{(p,p)}$

Usually only the kernel function *K(u,v)* is defined explicitly, and the feature mapping ϕ is defined implicitly. Not all functions can be written as dot product

⇒ Necessary and sufficient conditions for a kernel function in the finite case

- **Symmetry**: *K(u,v)=K(v,u)*
- **Cauchy-Schwarz Inequality**: $K(u,v)^2 \leq K(u,u)K(v,v)$
- Kernel Matrix **K** is **positive semi-definite** ($\mathbf{x}^T\mathbf{K}\mathbf{x} \geq 0$ for all $\mathbf{x} \neq \{0\}^p$)

$$\mathbf{K} = \left(K(\mathbf{x_i}, \mathbf{x_j}) \right)_{(i,j)=(1,1)}^{(n,n)}$$

In the infinite-dimensional case: *Mercer's Theorem*

Common kernel functions

polynomial kernel of degree *d*: $K(u,v)=(\langle u.v \rangle + c)^d$ (*linear kernel* if *d*=1)

Radial basis function (RBF): $K(u,v)=\exp(-\|u-v\|^2/c)$

Kernels may be used as background domain knowledge, but are quite opaque.

alex@seewald.at / alex.seewald.at

Sequential Minimal Optimization (SMO)

Many ways exist to solve the dual optimization problem iteratively. We will focus on one simple algorithm, Sequential Minimal Optimization (SMO).

- Start with $\alpha_i=0$ for all *i*. This ensures that $\Sigma\alpha_i y_i=0$ initially.
- Choose α_i , α_j arbitrarily (usually by heuristic to speed up convergence)
- The partial solution for α_i and α_j can be computed analytically:

$$\alpha_j^{new,unc} = \alpha_j^{old} + \frac{y_j\left((f(\mathbf{x_i})-y_i)-(f(\mathbf{x_j})-y_j)\right)}{K(\mathbf{x_i},\mathbf{x_i})+K(\mathbf{x_j},\mathbf{x_j})-2K(\mathbf{x_i},\mathbf{x_j})}$$

$$\alpha_j^{new} = \begin{cases} V & \text{if } \alpha_j^{new,unc} > V \\ \alpha_j^{new,unc} & \text{if } U \le \alpha_j^{new,unc} \le V \\ U & \text{if } \alpha_j^{new,unc} < U \end{cases}$$

$$\alpha_i^{new} = \alpha_i^{old} + y_i y_j (\alpha_j^{old} - \alpha_j^{new})$$

$$U = \begin{cases} \max(0, \alpha_j^{old} - \alpha_i^{old}) & \text{if } y_i \neq y_j \\ \max(0, \alpha_j^{old} + \alpha_i^{old} - \lambda) & \text{if } y_i = y_j \end{cases}$$

$$V = \begin{cases} \min(\lambda, \alpha_j^{old} - \alpha_i^{old} + \lambda) & \text{if } y_i \neq y_j \\ \min(\lambda, \alpha_j^{old} + \alpha_i^{old}) & \text{if } y_i = y_j \end{cases}$$

This brings us one step nearer to the solution by increasing L_D while maintaining the simpler constraints for the dual problem, i.e. $\Sigma\alpha_i y_i=0$

Repeat until convergence

Determine $\beta_0=y_i-f(\mathbf{x_i})$ (computing $f(\mathbf{x})$ with $\beta_0=0$) by averaging over all support vectors $0<\alpha_i<\lambda$ (implies $\xi_i=0$) for numerical stability.

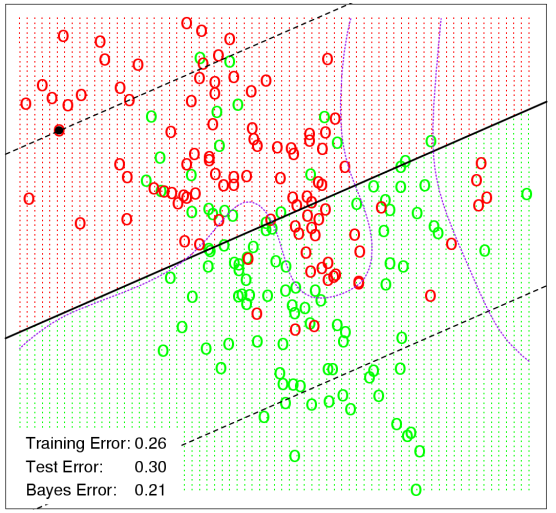
alex@seewald.at / alex.seewald.at

Complexity parameter λ

Influence of λ on SVM performance

Linear kernel ($\phi(x)=x$, $d=1$), $\lambda=0.01$

- Focusses more on data which is further away from the maximum margin hyperplane. Bigger margin reflects this behaviour. Error = 30.0%

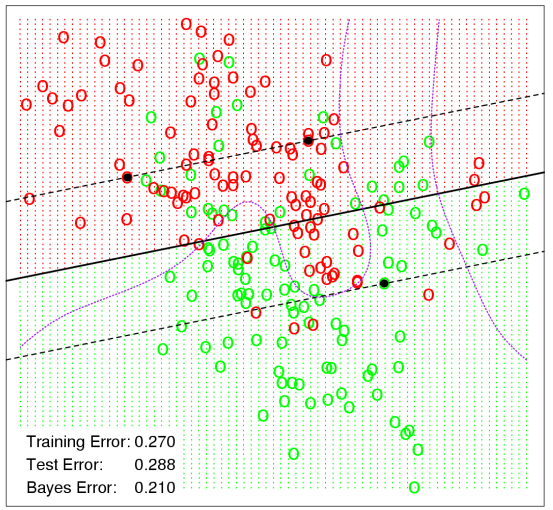


Linear kernel ($\phi(x)=x$, $d=1$), $\lambda=10000$

- Focusses more on data which is nearer to the maximum margin hyperplane. Smaller margin reflects this behaviour. Error = 28.8%

In both cases, all examples which are on the wrong side of the margin are given weight depending on their distance from the margin.

As we can see, the example is not well separable with just a linear kernel. Can we do better?



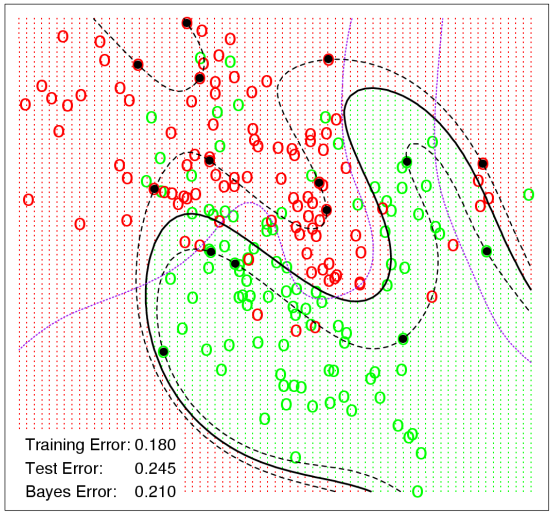
alex@seewald.at / alex.seewald.at

Non-linear Kernel Functions

Influence of Kernel on SVM performance

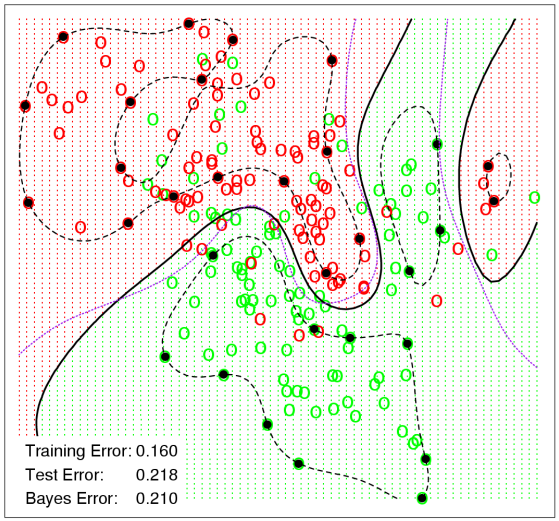
polynomial kernel of degree d=4, λ=1

- Non-linear decision boundary, small margin; slightly better generalization performance but tends to overshoot at the boundaries (a common problem of polynomials): Error = 24.5%



Gaussian (RBF) *kernel*, c=0.01, λ=1

- Non-linear decision boundary, larger margin; but almost optimal generalization performance. Error = 21.8% vs. optimal Bayes Error = 21.0%. This is probably due to the synthetic dataset which was generated by a mixture of Gaussian distributions.



alex@seewald.at / alex.seewald.at