Computational Statistics with Application to Bioinformatics

Prof. William H. Press
Spring Term, 2008
The University of Texas at Austin

Unit 18: Support Vector Machines (SVMs)

Unit 18: Support Vector Machines (SVMs) (Summary)

- Simplest case of SVM is linear separation by a fat plane
  - a quadratic programming problem in the space of features
  - its dual problem is in a space with dimension the number of data points
  - both problems readily solvable by standard algorithms
- Next allow “soft margin” approximate linear separation
  - gives an almost identical quadratic programming problem
  - there is now a free parameter λ (smoothing)
  - it is not the ROC parameter, though it does affect TPR vs. FPR
- The Kernel Trick
  - embedding in a high dimensional space makes linear separation much more powerful
  - this is easily accomplished in the dual space
  - in fact, you can “guess a kernel” and jump to effectively infinite dimensional space
  - you have to guess or search for the optimal parameters of your kernel
- Good SVM implementations (e.g., SVMlight) are freely available
  - we demonstrate by classifying yeast genes as mitochondrial or not

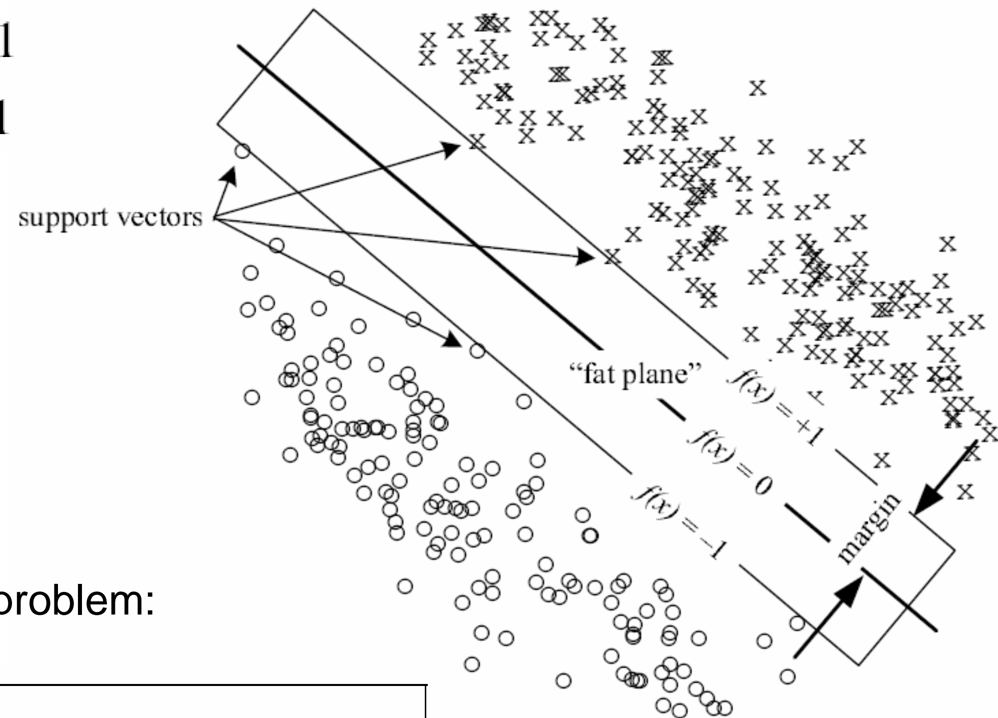
Support Vector Machine (SVM) is an algorithm for supervised binary classification

First, consider the case of data that is perfectly separable by a “fat plane”.
This is called “Maximum Margin SVM”.

$$\mathbf{w}\cdot\mathbf{x}_i + b \geq +1 \quad \text{when } y_i = +1$$
$$\mathbf{w}\cdot\mathbf{x}_i + b \leq -1 \quad \text{when } y_i = -1$$

$$2 \times \text{margin} = 2(\mathbf{w}\cdot\mathbf{w})^{-1/2}$$

i.e., project x to a single coordinate along w



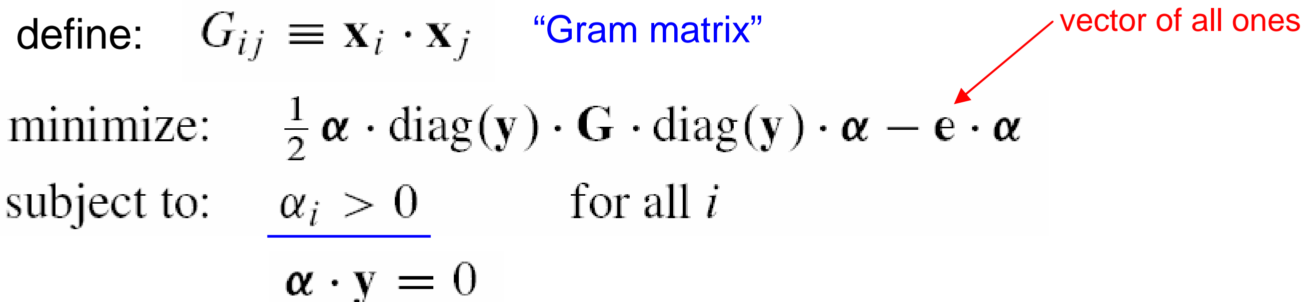
So we have a quadratic programming problem:

$$\begin{array}{ll} \text{minimize:} & \frac{1}{2}\mathbf{w}\cdot\mathbf{w} \\ \text{subject to:} & y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 \qquad i = 1,\ldots,m \end{array}$$

notice that this problem “lives” in a space with dimension the same as x (the feature space)

It turns out that every “primal” problem in quadratic programming (or more generally convex programming) has an equivalent “dual” problem [strong duality and Kuhn-Tucker theorems]

The dual problem for perfectly separable SVM is:



Notice that this dual problem lives in a space of dimension “number of data points”, and that, except for calculating G_{ij}, the dimension of the feature space is irrelevant.

This is going to let us move into infinite dimensional spaces!

Once you have found the minimizing α’s, you get the actual answer by

$$\hat{\mathbf{w}} = \sum_i \hat{\alpha}_i y_i \mathbf{x}_i \qquad \hat{b} = \sum_i \hat{\alpha}_i (y_i - \hat{\mathbf{w}} \cdot \mathbf{x}_i) \Big/ \sum_i \alpha_i$$

And the classifier is the sign of

$$f(\mathbf{x}) = \hat{\mathbf{w}} \cdot \mathbf{x} + \hat{b}$$

The "1-Norm Soft-Margin SVM" and Its Dual

We state, but gloss over the details: To go from "maximum margin" to "soft margin" (allowing imperfect separations), the only equation that changes is

$$\text{subject to:} \quad 0 \le \alpha_i \le \lambda \qquad \text{for all } i$$

where λ is a new "softness parameter" (inverse penalty for unseparated points). See NR3.

$$\hat{\alpha}_i = 0 \iff \text{data point } i \text{ on correct side of fat plane}$$

$$0 < \hat{\alpha}_i < \lambda \iff \text{data point } i \text{ exactly on fat plane boundary (a support vector)}$$

$$\hat{\alpha}_i = \lambda \iff \text{data point } i \text{ inside, or on wrong side, of fat plane}$$

You get to choose λ. It is a smoothing parameter:

$\lambda \to 0$ Smooth classifier, really fat fat-plane, less accurate on training data, but possibly more robust on new data

$\lambda \to \infty$ Sharp classifier, thinner fat-plane, more accurate on training data, but possibly less robust on new data

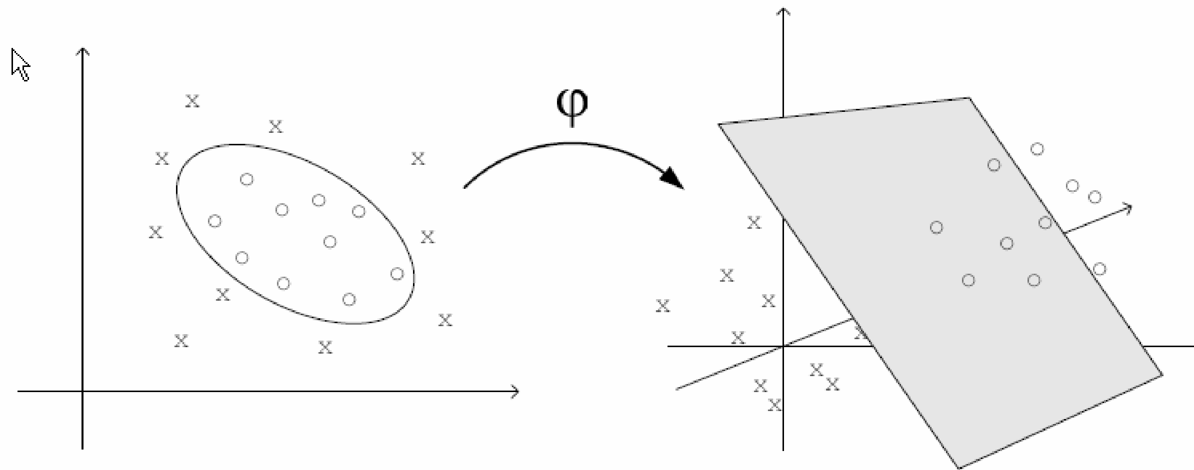
TPR and FPR will vary with λ, but it is not the ROC parameter, which would vary b.
You should view each choice of λ as being a different classifier, with its own ROC curve.
There is not always one that dominates.

“The Kernel Trick”

This is the real power of SVM. Otherwise it would just be linear fiddling around.

Imagine an embedding function into a much higher dimensional space:

$$\mathbf{x} \quad (n\text{-dimensional}) \quad \longrightarrow \quad \boldsymbol{\varphi}(\mathbf{x}) \quad (N\text{-dimensional})$$



The point is that very complicated separations in n-space can be represented as linear separations in N-space. A simple example is:

$$(x_0, x_1) \quad \xrightarrow{\varphi} \quad (x_0^2, x_0 x_1, x_1^2, x_0, x_1)$$

which would allow all quadratic separations.

In the embedding space, the (dual) problem to be solved is

$$\begin{aligned} &\text{minimize:} && \tfrac{1}{2}\,\boldsymbol{\alpha}\cdot\operatorname{diag}(\mathbf{y})\cdot\mathbf{K}\cdot\operatorname{diag}(\mathbf{y})\cdot\boldsymbol{\alpha} - \mathbf{e}\cdot\boldsymbol{\alpha} \\ &\text{subject to:} && 0 \le \alpha_i \le \lambda \qquad \text{for all } i \\ & && \alpha\cdot\mathbf{y} = 0 \end{aligned}$$

with $K_{ij} \equiv K(\mathbf{x}_i, \mathbf{x}_j) \equiv \boldsymbol{\varphi}(\mathbf{x}_i)\cdot\boldsymbol{\varphi}(\mathbf{x}_j)$

Now the "kernel trick" is: instead of guessing an embedding ϕ, just guess a kernel **K** directly! This is breathtakingly audacious, but it often works!

Properties of kernels that could have come from an embedding, even if you don't know what the embedding is:

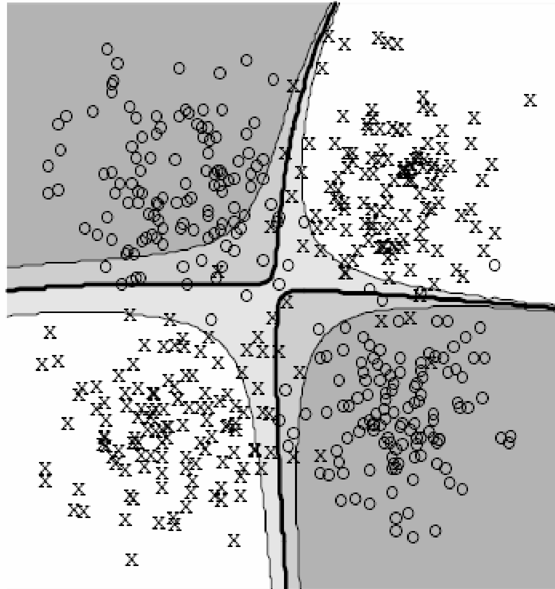
- $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j)$ must be symmetric in i and j and must have nonnegative eigenvalues (Mercer's theorem).
- Any multinomial combination of kernel functions is a kernel function. That is, you can freely combine kernel functions by multiplication, addition, and scaling by a constant.
- $K(\boldsymbol{\varphi}(\mathbf{x}_i), \boldsymbol{\varphi}(\mathbf{x}_j))$ is a kernel if $K(,)$ is one, for any $\boldsymbol{\varphi}$. This generalizes the original idea of the embedding space.
- $K(\mathbf{x}_i, \mathbf{x}_j) = g(\mathbf{x}_i)g(\mathbf{x}_j)$ is always a kernel, for any function g.

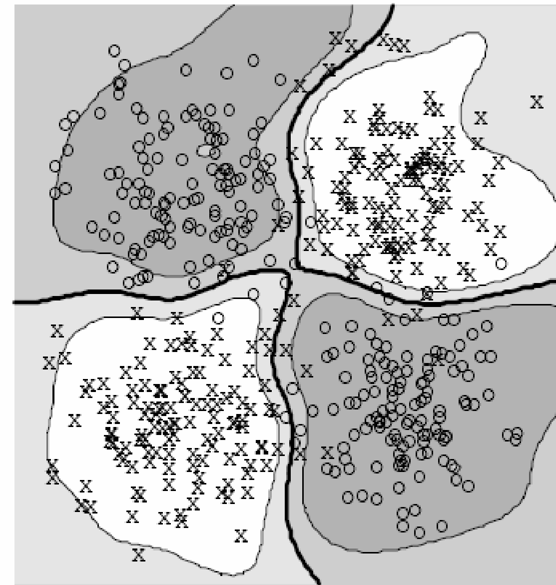
In practice, you rarely make up your own kernels, but rather use one or another standard, tried-and-true forms:

linear:	$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
power:	$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j)^d, \qquad 2 \le d \le 20 \text{ (say)}$
polynomial:	$K(\mathbf{x}_i, \mathbf{x}_j) = (a\,\mathbf{x}_i \cdot \mathbf{x}_j + b)^d$
sigmoid:	$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(a\,\mathbf{x}_i \cdot \mathbf{x}_j + b)$
Gaussian radial basis function:	$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{1}{2}\lvert\mathbf{x}_i - \mathbf{x}_j\rvert^2/\sigma^2)$

The kernel trick, by the way, is applicable to various other classification algorithms, including PCA. The field is called “kernel-based learning algorithms”.

Example: Learn a 2-d classification on data with no linear approximation



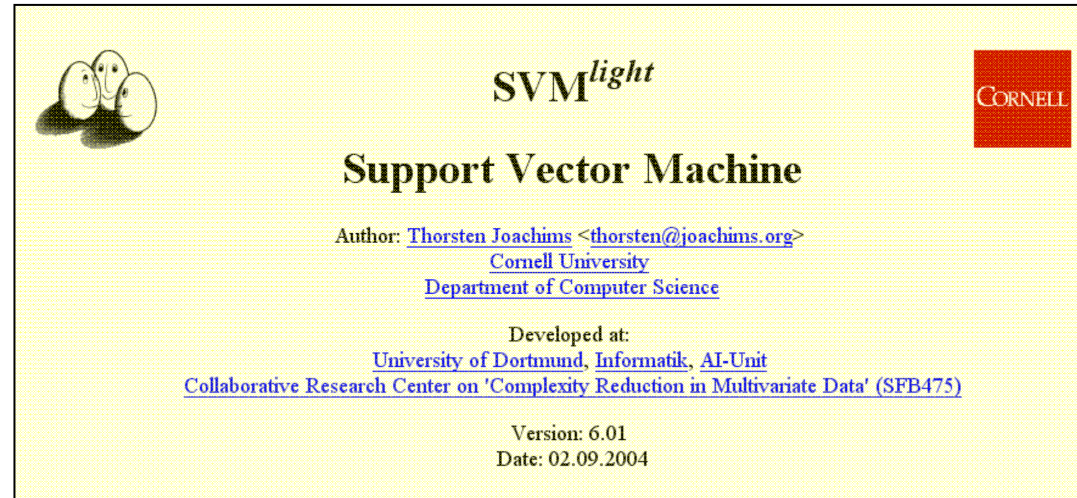


polynomial d=8 Gaussian radial basis function

When you pick a kernel, you have to play around to find good parameters, and also for choice of λ, the softness parameter. SVM is fundamentally heuristic.

Always divide your data into disjoint training and testing sets for this, or else you will get bitten by "overtraining".

Although NR3 has a “toy” SVM code, this is a case where you should use someone else’s well-optimized code. For SVM, a good (free) starting point is

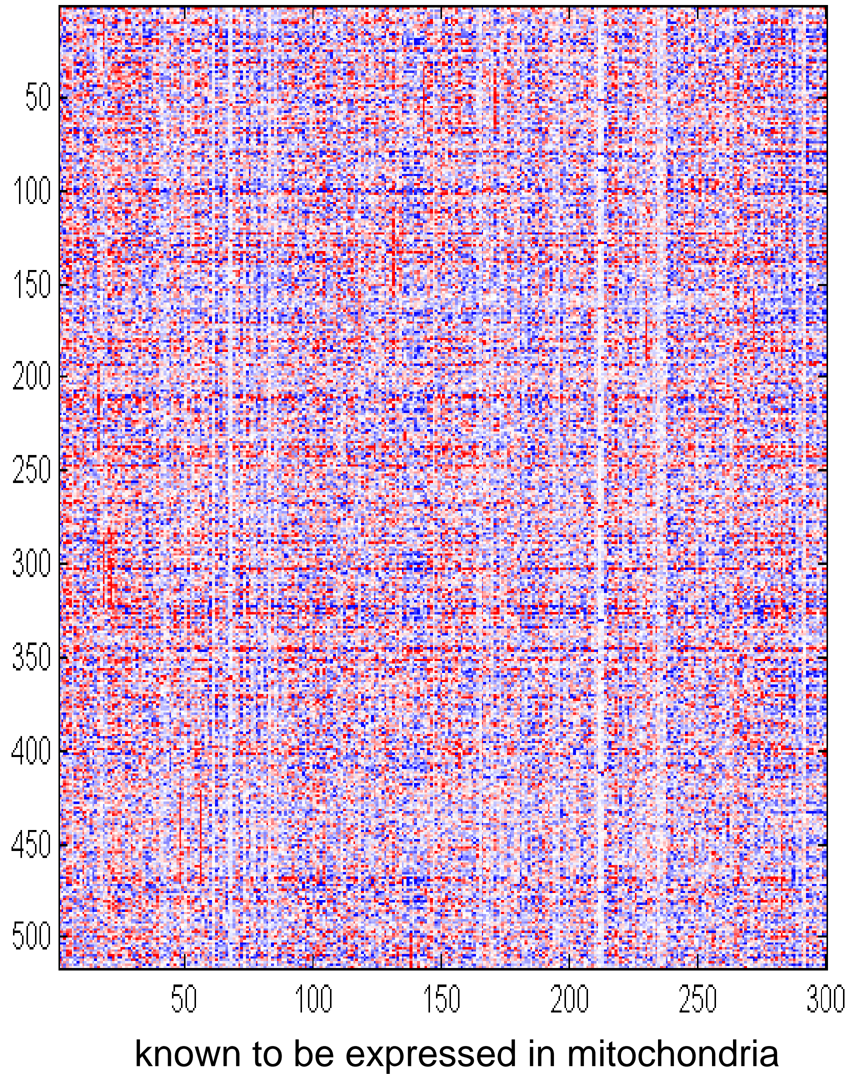



```
mytrain.txt:
+1 1:0.4 2:-1.5 3:-0.1 4:3.2 …
-1 1:-1.6 2:-0.5 3:-0.2 4:-1.2 …
+1 1:-0.9 3:-0.7 4:1.1 …
…
```

e.g., power kernel with d=2

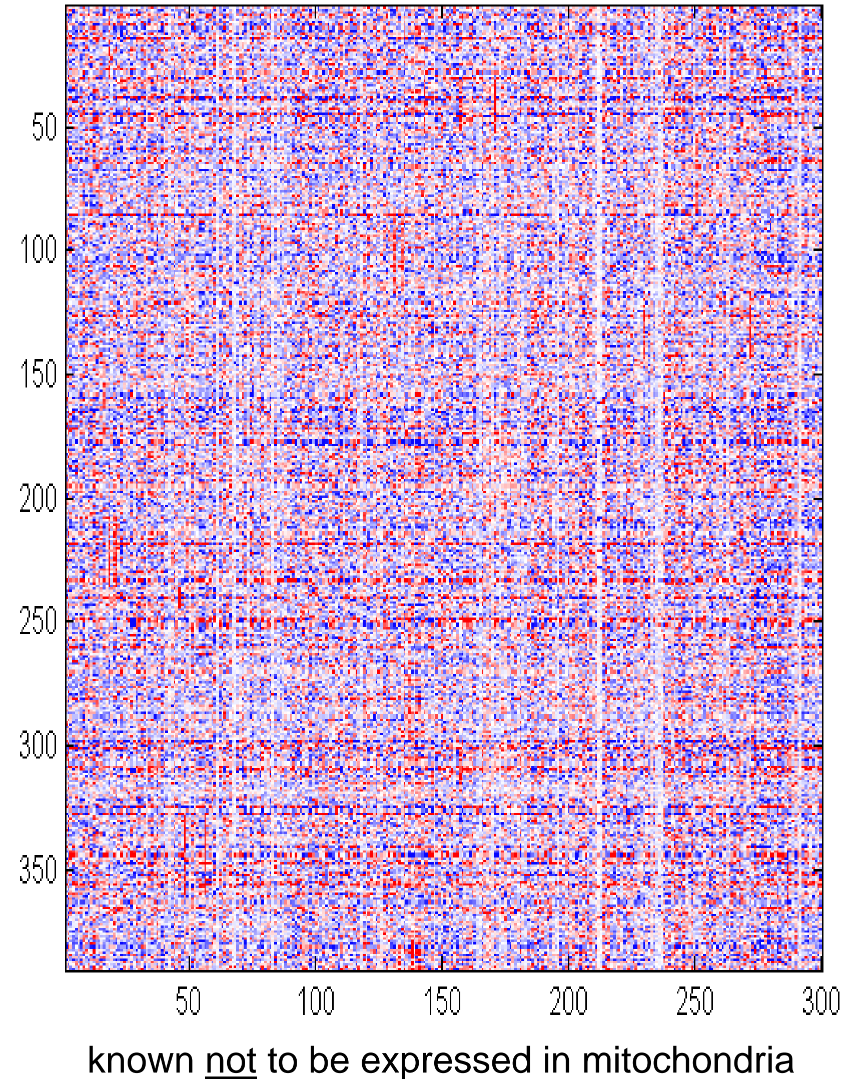
```
> svm_learn –t 1 –d 2 mytrain.txt mymodel
> svm_classify mytest.txt mymodel myanswer
```

Let's see if we can learn to predict yeast mitochondrial genes from gene expression profiles.



known to be expressed in mitochondria

svm_mito.txt on course web site



known <u>not</u> to be expressed in mitochondria

svm_nonmito.txt

linear kernel

svm_train.txt and svm_test.txt on course web site

```
svm_learn -t 0 svm_train.txt svm_model
Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..700..800..900..1000
..1100..1200..1300..1400..1500..1600..1700..1800..1900..2000..2100..2200..2300..
2400..2500..2600..2700..2800..2900..3000..OK. (3034 examples read)
Setting default regularization parameter C=0.0048
Optimizing...............................................................
....................done. (2091 iterations)
```

```
svm_classify svm_test.txt svm_model svm_output
Reading model...OK. (789 support vectors read)
Classifying test examples..100..200..300..400..500..600..700..800..900..1000..done
Runtime (without IO) in cpu-seconds: 0.00
Accuracy on test set: 89.92% (910 correct, 102 incorrect, 1012 total)
Precision/recall on test set: 79.31%/33.82%
```

SVMlight parameterizes by ACC, precision, recall, and N! (Why?)

prediction \ actual		
	TP	FP
	FN	TN

prediction \ actual		
	46	12
	89	865

Derive yet another confusion matrix transformation, now from (N, accuracy, precision, recall) to (TP,FN,FP,TN):

```
In[1]:= eqs = {pr == tp / (tp + fp), re == tp / (tp + fn),
          ac == (tp + tn) / (tp + tn + fp + fn), n == tp + tn + fp + fn}
```

Out[1]= $\left\{pr == \frac{tp}{fp+tp},\ re == \frac{tp}{fn+tp},\ ac == \frac{tn+tp}{fn+fp+tn+tp},\ n == fn+fp+tn+tp\right\}$

```
In[4]:= soln = FullSimplify[Solve[eqs, {tp, fp, fn, tn}]]
```

Out[4]= $\left\{\left\{tn \to \frac{n\,(-pr\,re + ac\,(pr + re - pr\,re))}{pr + re - 2\,pr\,re},\ fp \to -\frac{n\,(-1 + ac + pr - ac\,pr)\,re}{pr + re - 2\,pr\,re},\ tp \to -\frac{(-1 + ac)\,n\,pr\,re}{pr + re - 2\,pr\,re},\ fn \to -\frac{n\,pr\,(-1 + ac + re - ac\,re)}{pr + re - 2\,pr\,re}\right\}\right\}$

```
In[5]:= soln /. {n → 1012, ac → .90, pr → .79, re → .34}

Out[5]= {{tn → 864.946, fp → 12.1891, tp → 45.8541, fn → 89.0109}}

In[6]:= soln /. {n → 1012, ac → .92, pr → .92, re → .40}

Out[6]= {{tn → 880.024, fp → 4.43616, tp → 51.0159, fn → 76.5238}}
```

prediction \ actual		
	TP	FP
	FN	TN

Gaussian radial basis function

```
svm_learn -t 2 -g .001 svm_train.txt svm_model
Scanning examples...done
Reading examples into memory...100..200..300..400..500..600..700..800..900..1000
..1100..1200..1300..1400..1500..1600..1700..1800..1900..2000..2100..2200..2300..
2400..2500..2600..2700..2800..2900..3000..OK. (3034 examples read)
Setting default regularization parameter C=2.8406
Optimizing.............................................................
.................... Checking optimality of inactive variables...done.
```

```
svm_classify svm_test.txt svm_model svm_output
Reading model...OK. (901 support vectors read)
Classifying test examples..100..200..300..400..500..600..700..800..900..1000..done
Runtime (without IO) in cpu-seconds: 0.56
Accuracy on test set: 91.50% (926 correct, 86 incorrect, 1012 total)
Precision/recall on test set: 91.67%/40.44%
```

prediction \ actual		
	TP	FP
	FN	TN

prediction \ actual		
	51	4
	76	880

- SVM is not a panacea for all binary classification problems
  - actually, I had to play around to find a convincing example
  - mitochondrial genes worked, but nuclear genes didn't!
- It works best when you have about equal numbers of + and – in the training set
  - but there are weighting schemes to try to correct for this
  - mitochondrial genes worked well, even at 1:10 in training set
- Although you have to guess parameters, you can usually try powers of 10 values to bracket them
  - can automate a systematic search if you need to
- Every once in a while, SVM <u>really</u> works well on a problem!
- Also, contains many key ideas of convex programming
  - also used in interior-point methods for linear programming
- There are lots of generalizations of SVM