An Idiot's guide to Support vector machines (SVMs)

R. Berwick, Village Idiot

SVMs: A New Generation of Learning Algorithms

- Pre 1980:
  - Almost all learning methods learned linear decision surfaces.
  - Linear learning methods have nice theoretical properties
- 1980's
  - Decision trees and NNs allowed efficient learning of non-linear decision surfaces
  - Little theoretical basis and all suffer from local minima
- 1990's
  - Efficient learning algorithms for non-linear functions based on computational learning theory developed
  - Nice theoretical properties.

Key Ideas

- Two independent developments within last decade
  - New efficient separability of non-linear regions that use "kernel functions" : generalization of 'similarity' to new kinds of similarity measures based on dot products
  - Use of quadratic optimization problem to avoid 'local minimum' issues with neural nets
  - The resulting learning algorithm is an optimization algorithm rather than a greedy search

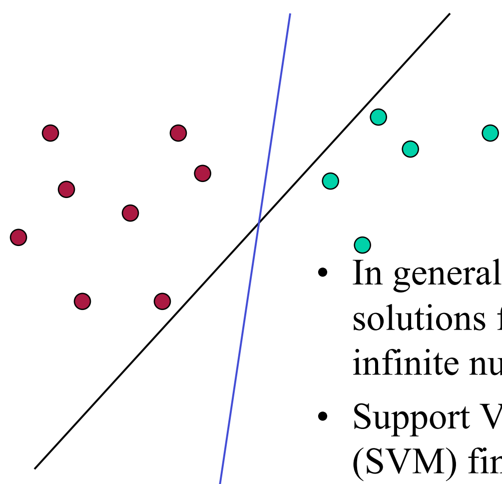
Organization

- Basic idea of support vector machines: just like 1-layer or multi-layer neural nets
  - Optimal hyperplane for linearly separable patterns
  - Extend to patterns that are not linearly separable by transformations of original data to map into new space – the <u>Kernel function</u>
- SVM algorithm for pattern recognition

Support Vectors

- Support vectors are the data points that lie closest to the decision surface (or hyperplane)
- They are the data points most difficult to classify
- They have direct bearing on the optimum location of the decision surface
- We can show that the optimal hyperplane stems from the function class with the lowest "capacity"= # of independent features/parameters we can twiddle [note this is 'extra' material not covered in the lectures… you don't have to know this]

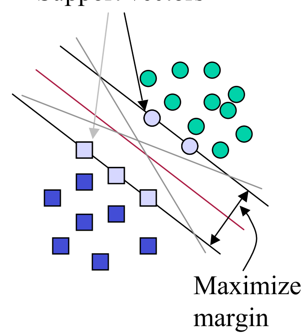
Recall from 1-layer nets : Which Separating Hyperplane?



- In general, lots of possible solutions for a,b,c (an infinite number!)
- Support Vector Machine (SVM) finds an optimal solution

Support Vector Machine (SVM)

- SVMs <u>maximize</u> the margin (Winston terminology: the ‘street’) around the separating hyperplane.
- The decision function is fully specified by a (usually very small) <u>subset</u> of training samples, the <u>support vectors.</u>
- This becomes a Quadratic programming problem that is easy to solve by standard methods



Separation by Hyperplanes

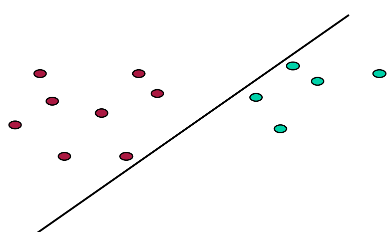
- Assume linear separability for now (we will relax this later)
- in 2 dimensions, can separate by a line
  - in higher dimensions, need hyperplanes

General input/output for SVMs just like for neural nets, but for one important addition…

Input: set of (input, output) training pair samples; call the input sample features $x_1, x_2...x_n$, and the output result y. Typically, there can be lots of input features x_i.

Output: set of weights **w** (or w_i), one for each feature, whose linear combination predicts the value of y. (So far, just like neural nets…)
Important difference: we use the optimization of maximizing the margin ('street width') to reduce the number of weights that are nonzero to just a few that correspond to the important features that 'matter' in deciding the separating line(hyperplane)…these nonzero weights correspond to the support vectors (because they 'support' the separating hyperplane)

2-D Case



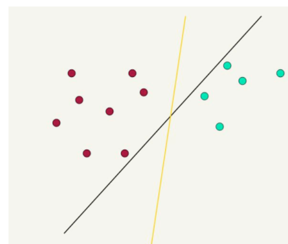
Find a,b,c, such that

$ax + by \geq c$ for red points

$ax + by \leq$ (or $<$) c for green points.

<u>Which</u> Hyperplane to pick?

- Lots of possible solutions for *a,b,c.*
- Some methods find a separating hyperplane, but not the optimal one (e.g., neural net)
- But: <u>Which</u> points should influence optimality?
  - All points?
    - Linear regression
    - Neural nets
  - Or only "difficult points" close to decision boundary
    - Support vector machines

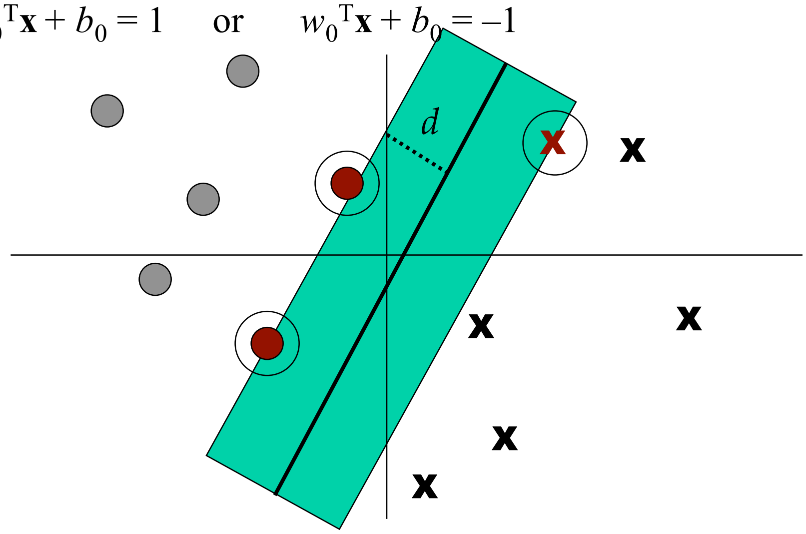


Support Vectors again for linearly separable case

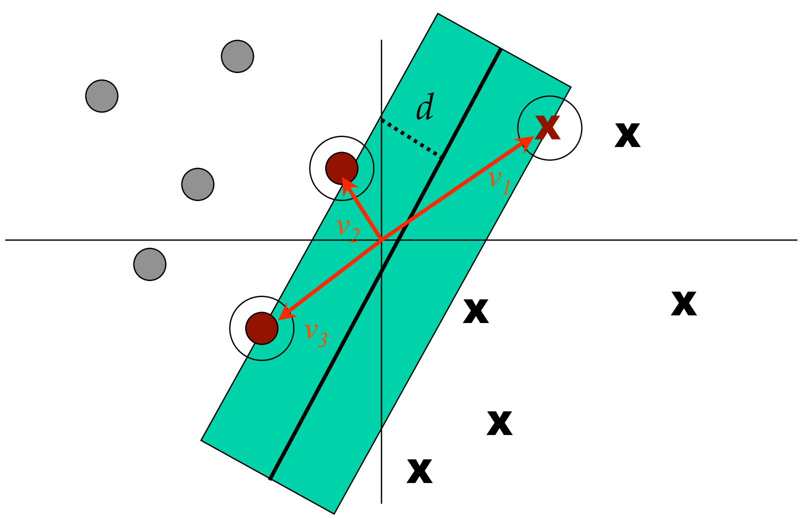
- Support vectors are the elements of the training set that would <u>change the position</u> of the dividing hyperplane if removed.
- Support vectors are the <u>critical</u> elements of the training set
- The problem of finding the optimal hyper plane is an optimization problem and can be solved by optimization techniques (we use Lagrange multipliers to get this problem into a form that can be solved analytically).

Support Vectors: Input vectors that just touch the boundary of the margin (street) – circled below, there are 3 of them (or, rather, the 'tips' of the vectors

$w_0^T\mathbf{x} + b_0 = 1$ or $w_0^T\mathbf{x} + b_0 = -1$



Here, we have shown the actual support vectors, v_1, v_2, v_3, instead of just the 3 circled points at the tail ends of the support vectors. d denotes 1/2 of the street 'width'

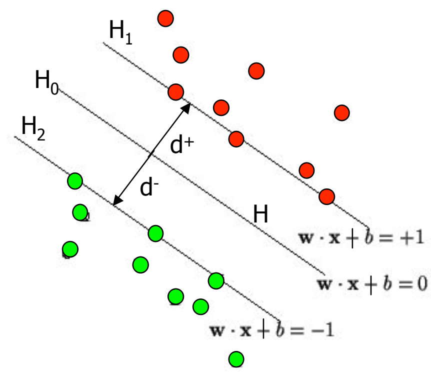


Definitions

Define the hyperplanes H such that:

$w \bullet x_i + b \geq +1$ when $y_i = +1$

$w \bullet x_i + b \leq -1$ when $y_i = -1$



H_1 and H_2 are the planes:

H_1: $w \bullet x_i + b = +1$

H_2: $w \bullet x_i + b = -1$

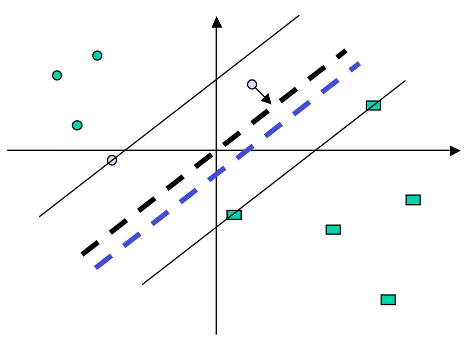
The points on the planes H_1 and H_2 are the tips of the <u>Support Vectors</u>

The plane H_0 is the median in between, where $w \bullet x_i + b = 0$

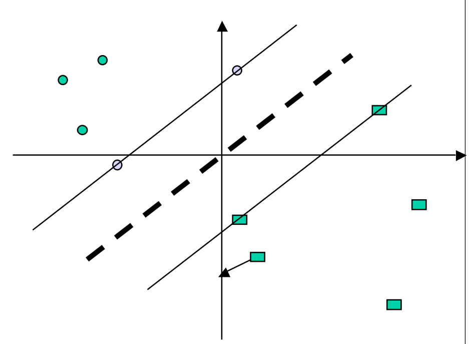
d+ = the shortest distance to the closest positive point

d- = the shortest distance to the closest negative point

The margin (gutter) of a separating hyperplane is d+ + d–.



Moving a support vector moves the decision boundary

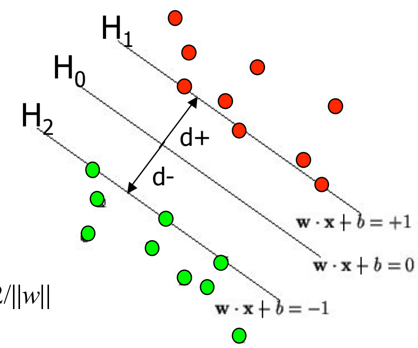


Moving the other vectors has no effect

The optimization algorithm to generate the weights proceeds in such a way that only the support vectors determine the weights and thus the boundary

Maximizing the margin (aka street width)

We want a classifier (linear separator) with as big a margin as possible.



Recall the distance from a point(x_0,y_0) to a line: $Ax+By+c = 0$ is: $|Ax_0 + By_0 + c|/\text{sqrt}(A^2+B^2)$, so,
The distance between H_0 and H_1 is then:
$|w \bullet x+b|/||w||=1/||w||$, so
The total distance between H_1 and H_2 is thus: $2/||w||$

In order to maximize the margin, we thus need to minimize $||w||$. With the condition that there are no datapoints between H_1 and H_2:

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

Can be combined into: $\mathbf{y_i(x_i \bullet w) \geq 1}$

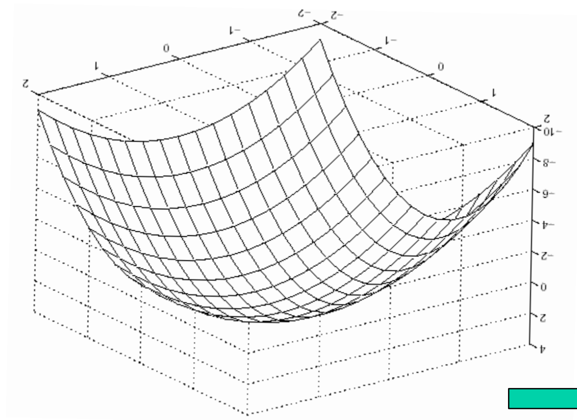
We now must solve a quadratic programming problem

- Problem is: minimize **$||\mathbf{w}||$, s.t.** discrimination boundary is obeyed, i.e., min $f(x)$ s.t. $g(x)$=0, which we can rewrite as:
  min f: $\frac{1}{2} ||w||^2$ (Note this is a quadratic function)
  s.t. g: $y_\mathbf{i}(\mathbf{w} \bullet \mathbf{x}_i)-\mathbf{b} = \mathbf{1}$ or $[y_\mathbf{i}(\mathbf{w} \bullet \mathbf{x}_i)-\mathbf{b}] - \mathbf{1} = \mathbf{0}$

This is a **constrained optimization problem**

It can be solved by the Lagrangian multipler method

Because it is quadratic, the surface is a paraboloid, with just a single global minimum (thus avoiding a problem we had with neural nets!)

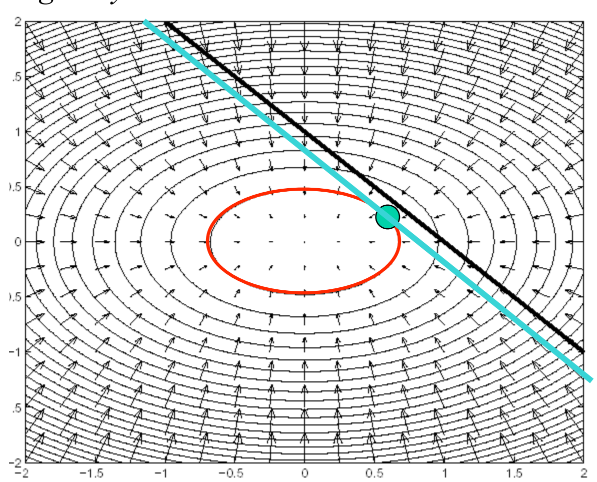


flatten

Example: paraboloid $2+x^2+2y^2$ s.t. $x+y=1$

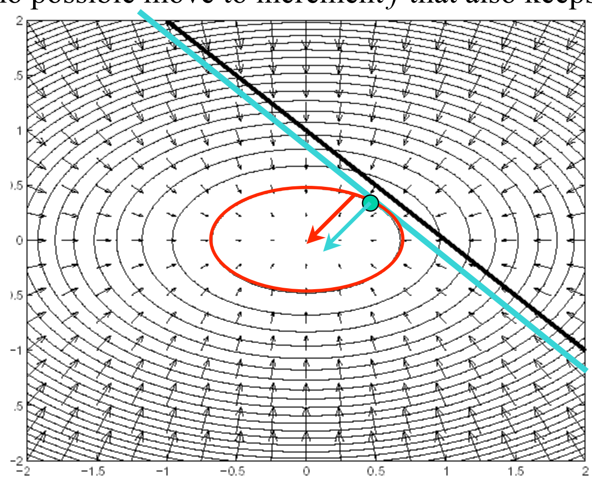
<u>Intuition:</u> find intersection of <u>two</u> functions *f, g* at a tangent point (intersection = <u>both</u> constraints satisfied; tangent = derivative is 0); this will be a min (or max) for *f* s.t. the contraint *g* is satisfied

Flattened paraboloid *f*: $2x^2+2y^2=0$ *with* superimposed constraint *g*: $x+y = 1$



Minimize when the constraint line *g* (shown in green) is <u>tangent</u> to the inner ellipse contour linez of *f* (shown in red) – note direction of gradient arrows.

flattened paraboloid *f*: $2+x^2+2y^2=0$ *with* superimposed constraint *g*: $x+y = 1$; at tangent solution *p*, gradient vectors of *f,g* are <u>parallel</u> (no possible move to increment *f* that also keeps you in region *g*)



Minimize when the constraint line *g* is <u>tangent</u> to the inner ellipse contour line of *f*

Two constraints

1. Parallel normal constraint (= gradient constraint on *f, g* s.t. solution is a max, or a min)
2. *g*(x)=0 (solution is on the constraint line as well)

We now recast these by combining *f, g* as the new <u>Lagrangian</u> function by introducing new 'slack variables' denoted *a* or (more usually, denoted α in the literature)

Redescribing these conditions

- Want to look for solution point *p* where

$$\nabla f(p) = \nabla \lambda g(p)$$
$$g(x) = 0$$

- Or, combining these two as the *Langrangian L* & requiring derivative of *L* be zero:

$$L(x,a) = f(x) - ag(x)$$
$$\nabla(x,a) = 0$$

At a solution *p*

- The the constraint line *g* and the contour lines of *f* must be <u>tangent</u>
- If they are tangent, their gradient vectors (perpendiculars) are <u>parallel</u>
- Gradient of *g* must be 0 – i.e., steepest ascent & so perpendicular to *f*
- Gradient of *f* must also be in the same direction as *g*

How Langrangian solves constrained optimization

$$L(x,a) = f(x) - ag(x) \text{ where}$$
$$\nabla(x,a) = 0$$

Partial derivatives wrt x recover the parallel normal constraint

Partial derivatives wrt λ recover the $g(x,y)$=0

In general, $$L(x,a) = f(x) + \sum_i a_i g_i(x)$$

In general

Gradient min of f

constraint condition g

$L(x,a) = f(x) + \sum_i a_i g_i(x)$ a function of $n+m$ variables

n for the x's, m for the a. Differentiating gives $n+m$ equations, each set to 0. The n eqns differentiated wrt each x_i give the gradient conditions; the m eqns differentiated wrt each a_i recover the constraints g_i

In our case, $f(x)$: ½|| **w**||^2 ; $g(x)$: $y_i(\mathbf{w}\bullet x_i + b) - 1 = 0$ so Lagrangian is:

$$\min L = ½\|\mathbf{w}\|^2 - \Sigma a_i[y_i(\mathbf{w}\bullet x_i + \mathrm{b}) - 1] \text{ wrt } \mathbf{w}, b$$

We expand the last to get the following L form:

$$\min L = ½\|\mathbf{w}\|^2 - \Sigma a_i y_i(\mathbf{w}\bullet x_i + \mathrm{b}) + \Sigma a_i \text{ wrt } \mathbf{w}, b$$

Lagrangian Formulation

- So in the SVM problem the Lagrangian is

$$\min L_P = \tfrac{1}{2}\left\|\mathbf{w}\right\|^2 - \sum_{i=1}^{l} a_i y_i \left(\mathbf{x}_i \cdot \mathbf{w} + b\right) + \sum_{i=1}^{l} a_i$$

s.t. $\forall i, a_i \geq 0$ where l is the # of training points

- From the property that the derivatives at min = 0 we get:

$$\frac{\partial L_P}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{l} a_i y_i \mathbf{x}_i = 0$$

$$\frac{\partial L_P}{\partial b} = \sum_{i=1}^{l} a_i y_i = 0 \quad \text{so}$$

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x}_i, \quad \sum_{i=1}^{l} a_i y_i = 0$$

What's with this L_p business?

- This indicates that this is the <u>primal</u> form of the optimization problem
- We will actually solve the optimization problem by now solving for the <u>dual</u> of this original problem
- What is this <u>dual</u> formulation?

The Lagrangian Dual Problem: instead of minimizing over **w**, *b*, subject to constraints involving *a*'s, we can maximize over *a* (the dual variable) subject to the relations obtained previously for **w** and *b*

Our solution must satisfy these two relations:

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x}_i, \quad \sum_{i=1}^{l} a_i y_i = 0$$

By substituting for **w** and *b* back in the original eqn we can get rid of the dependence on **w** and *b*.

Note first that we already now have our answer for what the weights **w** must be: they are a linear combination of the training inputs and the training outputs, x_i and y_i and the values of *a*. We will now solve for the *a*'s by differentiating the dual problem wrt *a*, and setting it to zero. Most of the *a*'s will turn out to have the value zero. The non-zero *a*'s will correspond to the support vectors

Primal problem:

$$\min L_P = \tfrac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{l} a_i y_i \left(\mathbf{x}_i \cdot \mathbf{w} + b\right) + \sum_{i=1}^{l} a_i$$

s.t. $\forall i \; a_i \geq 0$

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x}_i, \quad \sum_{i=1}^{l} a_i y_i = 0$$

Dual problem:

$$\max L_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l} a_i a_j y_i y_j \left(\mathbf{x}_i \cdot \mathbf{x}_j\right)$$

s.t. $\sum_{i=1}^{l} a_i y_i = 0 \;\&\; a_i \geq 0$

(note that we have removed the dependence on **w** and *b*)

The Dual problem

- Kuhn-Tucker theorem: the solution we find here will be the same as the solution to the original problem
- Q: But why are we doing this???? (why not just solve the original problem????)
- Ans: Because this will let us solve the problem by computing the just the inner products of x_i, x_j (which will be very important later on when we want to solve non-linearly separable classification problems)

The Dual Problem

Dual problem:

$$\max L_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l} a_i a_j y_i y_j \left(\mathbf{x}_i \cdot \mathbf{x}_j\right)$$

$$\text{s.t. } \sum_{i=1}^{l} a_i y_i = 0 \text{ \& } a_i \geq 0$$

Notice that all we have are the dot products of x_i,x_j

If we take the derivative wrt a and set it equal to zero, we get the following solution, so we can solve for a_i:

$$\sum_{i=1}^{l} a_i y_i = 0$$

$$0 \leq a_i \leq C$$

Now knowing the a_i we can find the weights **w** for the maximal margin separating hyperplane:

$$\mathbf{w} = \sum_{i=1}^{l} a_i y_i \mathbf{x}_i$$

And now, after training and finding the **w** by this method, given an unknown point u measured on features x_i we can classify it by looking at the sign of:

$$f(x) = \mathbf{w} \bullet \mathbf{u} + b = (\sum_{i=1}^{l} a_i y_i \mathbf{x_i} \bullet \mathbf{u}) + b$$

Remember: most of the weights $\mathbf{w_i}$, i.e., the a's, will be zero
Only the support vectors (on the gutters or margin) will have nonzero weights or a's – this reduces the dimensionality of the solution

Inner products, similarity, and SVMs

Why should inner product kernels be involved in pattern recognition using SVMs, or at all?

– Intuition is that inner products provide some measure of 'similarity'

– Inner product in 2D between 2 vectors of unit length returns the cosine of the angle between them = how 'far apart' they are

e.g. $\mathbf{x} = [1, 0]^T$, $\mathbf{y} = [0, 1]^T$

i.e. if they are parallel their inner product is 1 (completely similar)

$$\mathbf{x}^T \mathbf{y} = \mathbf{x} \bullet \mathbf{y} = 1$$

If they are perpendicular (completely unlike) their inner product is 0 (so should not contribute to the correct classifier)

$$\mathbf{x}^T \bullet \mathbf{y} = \mathbf{x} \bullet \mathbf{y} = 0$$

Insight into inner products

Consider that we are trying to maximize the form:

$$L_D(a_i) = \sum_{i=1}^{l} a_i - \frac{1}{2}\sum_{i=1}^{l} a_i a_j y_i y_j \left(\mathbf{x}_i \cdot \mathbf{x}_j\right)$$

$$\text{s.t. } \sum_{i=1}^{l} a_i y_i = 0 \;\&\; a_i \geq 0$$

The claim is that this function will be <u>maximized</u> if we give nonzero values to a's that correspond to the support vectors, ie, those that 'matter' in fixing the maximum width margin ('street'). Well, consider what this looks like. Note first from the constraint condition that all the a's are positive. Now let's think about a few cases.

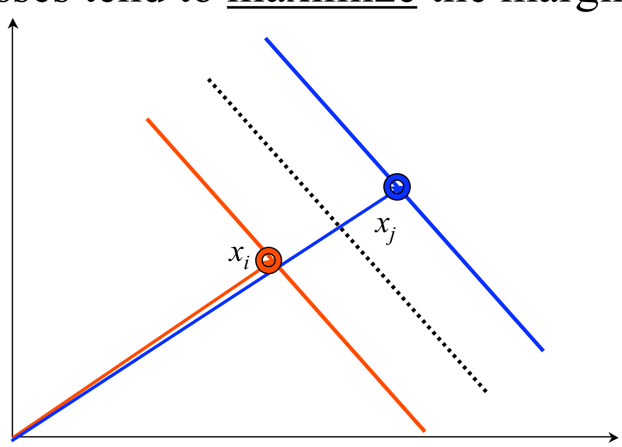
Case 1. If two features x_i , x_j are completely <u>dissimilar</u>, their dot product is 0, and they don't contribute to L.

Case 2. If two features x_i,x_j are completely <u>alike</u>, their dot product is 0. There are 2 subcases.

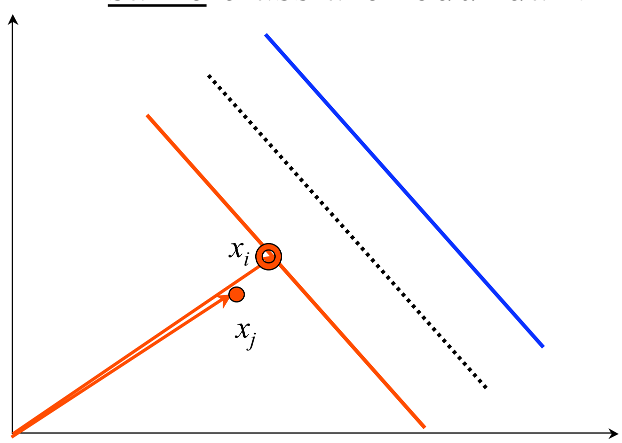
Subcase 1: both x_i, and x_j predict the <u>same</u> output value y_i (either +1 or –1). Then y_i x y_j is always 1, and the value of $a_i a_j y_i y_j x_i x_j$ will be positive. But this would <u>decrease</u> the value of L (since it would subtract from the first term sum). So, the algorithm downgrades similar feature vectors that make the <u>same</u> prediction.

Subcase 2: x_i, and x_j make <u>opposite</u> predictions about the output value y_i (ie, one is +1, the other –1), but are otherwise very closely similar: then the product $a_i a_j y_i y_j x_i x$ is <u>negative</u> and we are <u>subtracting</u> it, so this <u>adds</u> to the sum, maximizing it. This is precisely the examples we are looking for: the critical ones that tell the two classses apart.

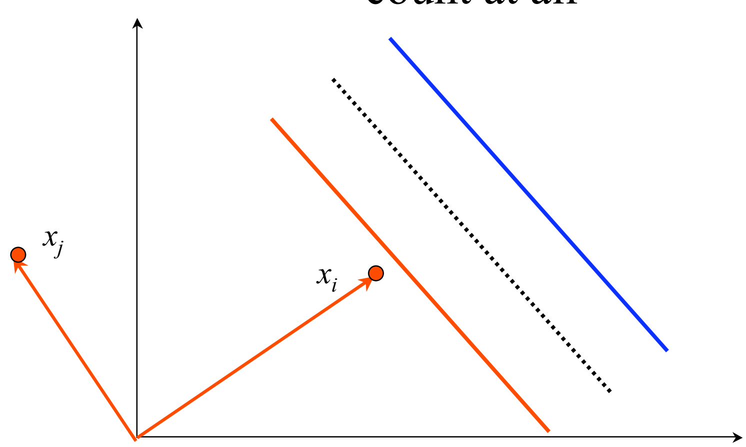
Insight into inner products, graphically: 2 very very similar x_i, x_j vectors that predict difft classes tend to <u>maximize</u> the margin width



2 vectors that are similar but predict the <u>same</u> class are redundant

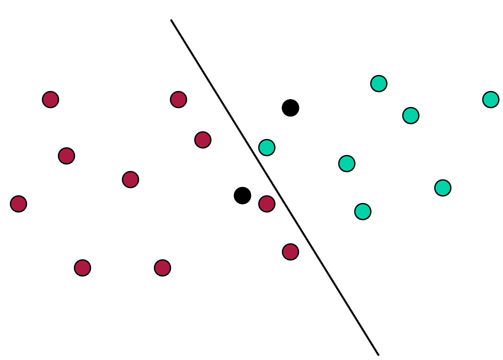


2 dissimilar (orthogonal) vectors don't count at all



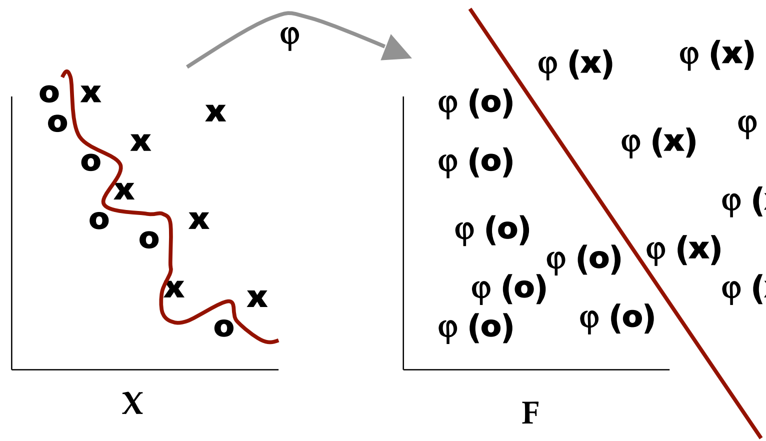
But…are we done???

Not Linearly Separable!



Find a line that penalizes points on “the wrong side”

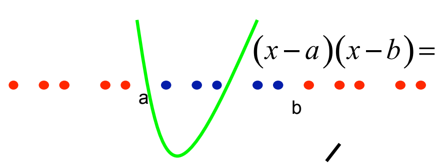
Transformation to separate



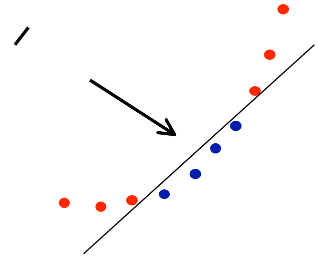
Non–Linear SVMs

- The idea is to gain linearly separation by mapping the data to a higher dimensional space
  - The following set can't be separated by a linear function, but can be separated by a quadratic one

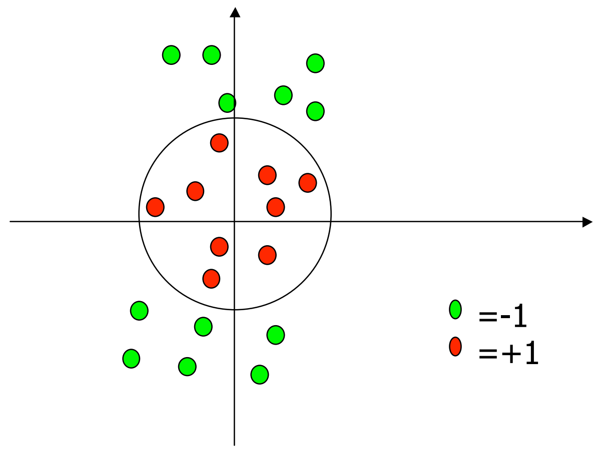
$$(x-a)(x-b)=x^2-(a+b)x+ab$$



  - So if we map $x \mapsto \{x^2, x\}$ we gain linear separation



Problems with linear SVM

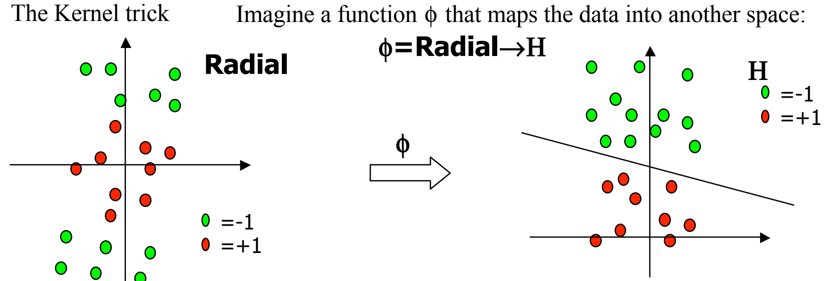


What if the decision function is not linear? What transform would separate these?

Ans: polar coordinates! Non-linear SVM

The Kernel trick

Imagine a function ϕ that maps the data into another space:



Remember the function we want to optimize: $L_d = \sum a_i - \frac{1}{2}\sum a_i a_j y_i y_j (x_i \bullet x_j)$ where $(x_i \bullet x_j)$ is the dot product of the two feature vectors. If we now transform to ϕ, instead of computing this dot product $(x_i \bullet x_j)$ we will have to compute $(\phi(x_i) \bullet \phi(x_j))$. But how can we do this? This is expensive and time consuming (suppose ϕ is a quartic polynomial… or worse, we don't know the function explicitly. Well, here is the neat thing:

If there is a "kernel function" K such that $K(x_i,x_j) = \phi(x_i) \bullet \phi(x_j)$, then we do not need to know or compute ϕ at all!! That is, the kernel function defines inner products in the transformed space. Or, it defines similarity in the transformed space.

Non-linear SVMs

So, the function we end up optimizing is:
$L_d = \sum a_i - \frac{1}{2}\sum a_i a_j y_i y_j K(x_i \bullet x_j)$,

Kernel example: The polynomial kernel
$K(xi,xj) = (x_i \bullet x_j + 1)^p$, where p is a tunable parameter
Note: Evaluating K only requires <u>one</u> addition and <u>one</u> exponentiation more than the original dot product

Examples for Non Linear SVMs

$$K(\mathbf{x},\mathbf{y}) = (\mathbf{x}\cdot\mathbf{y}+1)^p$$

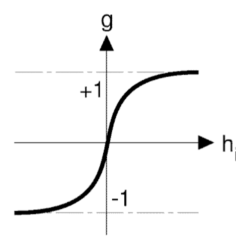
$$K(\mathbf{x},\mathbf{y}) = \exp\left\{-\|\mathbf{x}-\mathbf{y}\|^2 / 2\sigma^2\right\}$$

$$K(\mathbf{x},\mathbf{y}) = \tanh(\kappa\mathbf{x}\cdot\mathbf{y}-\delta)$$

1st is polynomial (includes x•x as special case)
2nd is radial basis function (gaussians)
3rd is sigmoid (neural net activation function)

We've already seen such <u>nonlinear</u> transforms…

- What is it???
- $\tanh(\beta_0 x^T x_i + \beta_1)$
- It's the sigmoid transform (for neural nets)
- So, SVMs <u>subsume</u> neural nets! (but w/o their problems…)



Inner Product Kernels

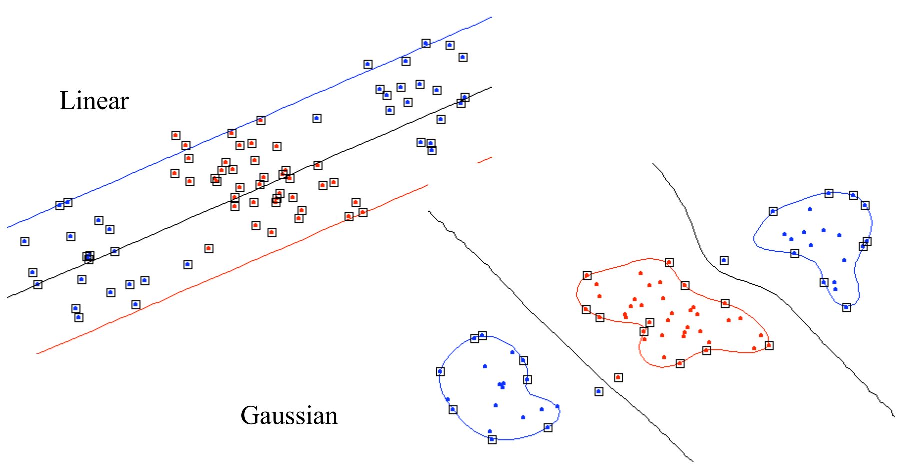
Type of Support Vector Machine	Inner Product Kernel $K(x,x_i)$, I = 1, 2, …, N	Usual inner product
Polynomial learning machine	$(x^T x_i + 1)^p$	Power *p* is specified *a priori* by the user
Radial-basis function (RBF)	$\exp(1/(2\sigma^2)\|\|x-x_i\|\|^2)$	The width σ^2 is specified *a priori*
Two layer neural net	$\tanh(\beta_0 x^T x_i + \beta_1)$	Actually works only for some values of β_0 and β_1

Kernels generalize the notion of 'inner product similarity'

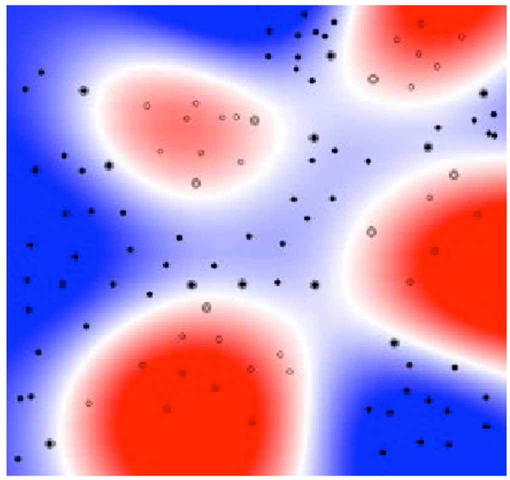
Note that one can define kernels over more than just vectors: strings, trees, structures, … in fact, just about anything

A very powerful idea: used in comparing DNA, protein structure, sentence structures, etc.

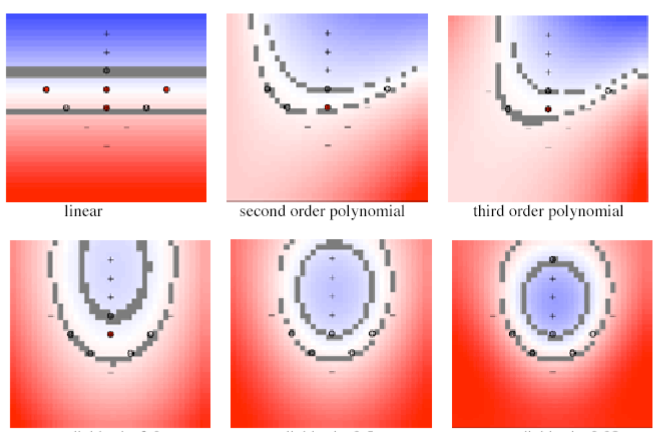
Examples for Non Linear SVMs 2 – Gaussian Kernel



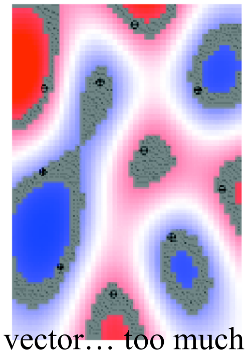
Nonlinear rbf kernel



Admiral's delight w/ difft kernel functions



Overfitting by SVM



Every point is a support vector… too much freedom to bend to fit the training data – no generalization.

In fact, SVMs have an 'automatic' way to avoid such issues, but we won't cover it here… see the book by Vapnik, 1995. (We add a penalty function for mistakes made after training by over-fitting: recall that if one over-fits, then one will tend to make errors on new data. This penalty fn can be put into the quadratic programming problem directly. You don't need to know this for this course.)