

Intro. to machine learning (CSI 5325)

Lecture 22: Support vector learning

Greg Hamerly

Spring 2008

Some content from Andrew Moore
<http://www.cs.cmu.edu/~awm/tutorials>.

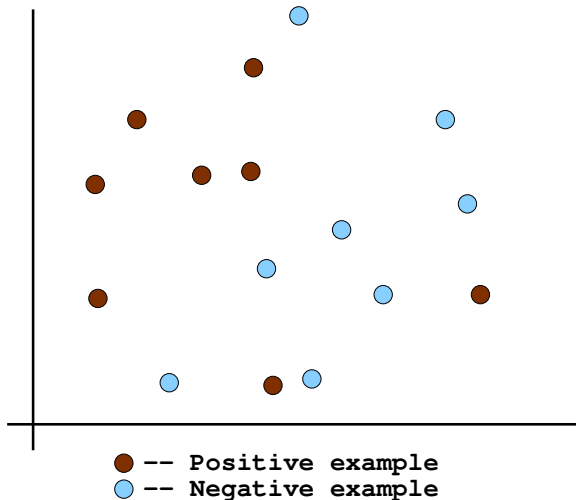
1 Non-separable case

2 Nonlinear SVMs

3 Kernel methods

Non-separable datasets

What if the data is not linearly separable?



Compensating for non-separable data: approach 1

Idea #1:

- minimize $\mathbf{w} \cdot \mathbf{w}$, while also minimizing number of training errors

Compensating for non-separable data: approach 1

Idea #1:

- minimize $\mathbf{w} \cdot \mathbf{w}$, while also minimizing number of training errors

Problem with this:

- Now we have to minimize two things, which makes the optimization ill-defined.

Compensating for non-separable data: approach 1.1

Idea #1.1:

- minimize $\mathbf{w} \cdot \mathbf{w} + C(\# \text{ train errors})$
- where C is a user-chosen tradeoff parameter

Compensating for non-separable data: approach 1.1

Idea #1.1:

- minimize $\mathbf{w} \cdot \mathbf{w} + C(\# \text{ train errors})$
- where C is a user-chosen tradeoff parameter

Problems with this:

- This can't be expressed as a QP problem.
- May be slow to implement.
- Doesn't distinguish between disastrous errors and near misses.

Compensating for non-separable data: approach 2

Idea #2:

- minimize

$$\mathbf{w} \cdot \mathbf{w} + C(\text{distance of misclassified points to their correct places})$$

- where C is a user-chosen tradeoff parameter

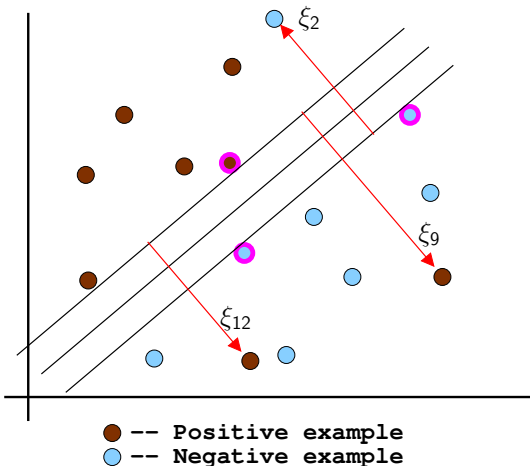
So how do we express this as a QP optimization?

- introduce 'slack variables' ξ_i into our constraints

Learning the maximum margin – with noise

Now we have slack variables ξ_i (one per example) to accomodate errors.

$\xi_i = 0$ if the point is correctly classified;
otherwise it is positive.



Learning the maximum margin – with noise

In the separable case, we had the following:

- minimize $\mathbf{w} \cdot \mathbf{w}$

Subject to n constraints:

- $\mathbf{w} \cdot \mathbf{x}_i - b \geq 1$ if $y_i = 1$
- $\mathbf{w} \cdot \mathbf{x}_i - b \leq -1$ if $y_i = -1$

Learning the maximum margin – with noise

In the separable case, we had the following:

- minimize $\mathbf{w} \cdot \mathbf{w}$

Subject to n constraints:

- $\mathbf{w} \cdot \mathbf{x}_i - b \geq 1$ if $y_i = 1$
- $\mathbf{w} \cdot \mathbf{x}_i - b \leq -1$ if $y_i = -1$

Now in the non-separable case, we have:

- minimize $\mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^n \xi_i$

Subject to $2n$ constraints:

- $\mathbf{w} \cdot \mathbf{x}_i - b \geq 1 - \xi_i$ if $y_i = 1$
- $\mathbf{w} \cdot \mathbf{x}_i - b \leq -1 + \xi_i$ if $y_i = -1$
- $\xi_i \geq 0$

An equivalent (dual) QP

The QP has a so-called 'dual' form which can be solved more efficiently.

- We want to **maximize** the dual form (recall we wanted to **minimize** the primal form).
- We also introduce Lagrange multipliers α_i (one for each example) which ensure the constraints.

$$\text{Maximize (over } \alpha_i): \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j (\mathbf{x}_i \cdot \mathbf{x}_j)$$

$$\text{Subject to:} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^n \alpha_i y_i = 0$$

Where did \mathbf{w} and b go? Where does C come from?

Finding \mathbf{w} and b from the dual's solution

After maximizing with respect to α_j , we can find:

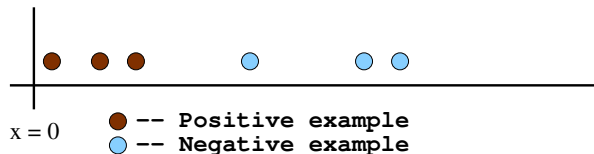
$$\begin{aligned}\mathbf{w} &= \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i \\ b &= y_i(1 - \xi_i) - \mathbf{x}_i \cdot \mathbf{w}_i\end{aligned}$$

Examples with $\alpha_j > 0$ will be the support vectors. (Remember that $0 \leq \alpha_j \leq C$.)

Thus, finding \mathbf{w} needs only sum over the support vectors.

1-dimensional data – simple

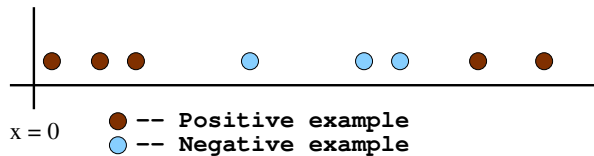
What would a linear SVM do with this data?



Where is the decision plane? Where are the support vectors?

1-dimensional data – problematic

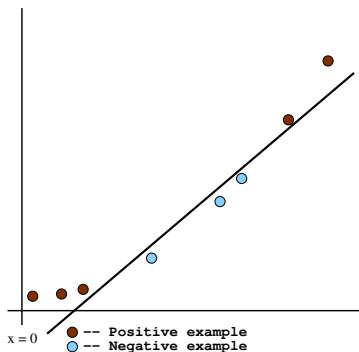
What about this data? Can a linear SVM separate it?



1-dimensional data – basis expansion

Take the data and apply a basis expansion to two dimensions:

$$x \rightarrow \Phi(x) = [x, x^2]$$



Now it's linearly separable! (In the expanded space...)

Also, \mathbf{w} now lives in the expanded space.

Common SVM basis expansions

- polynomial terms of \mathbf{x} of degrees 1 to p
- radial basis function of \mathbf{x}
- sigmoid function of \mathbf{x}

How do we define these? Hold that thought...

Consider quadratic basis functions

Consider $\Phi(\mathbf{x})$ that maps \mathbf{x} to another vector:

$\Phi(\mathbf{x})$ has how many terms?

$$\binom{d+2}{2} = \frac{(d+2)(d+1)}{2} \approx \frac{d^2}{2}$$

- 1 constant term
- d linear terms
- d self-multiplicative quadratic terms
- $d(d-1)/2$ quadratic cross terms

Why the factors of $\sqrt{2}$? We'll find out shortly.

$$\Phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ \dots \\ \sqrt{2}x_d \\ x_1^2 \\ x_2^2 \\ \dots \\ x_d^2 \\ \sqrt{2}x_1x_2 \\ \sqrt{2}x_1x_3 \\ \dots \\ \sqrt{2}x_1x_d \\ \sqrt{2}x_2x_3 \\ \dots \\ \sqrt{2}x_{d-1}x_d \end{bmatrix}$$

QP with basis functions

Maximize (over α_i):

$$\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$$

Subject to:

$$0 \leq \alpha_i \leq C, \quad \sum_{i=1}^n \alpha_i y_i = 0$$

Where we can again get the original parameters \mathbf{w} and b as:

$$\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \Phi(\mathbf{x}_i)$$

$$b = y_i(1 - \xi_i) - \Phi(\mathbf{x}_i) \cdot \mathbf{w}$$

(compare these with the linear formulation a few slides back)

QP with basis functions

Maximize (over α_i):

$$\sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_i^n \sum_j^n \alpha_i \alpha_j y_i y_j \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$$

How many dot products between $\Phi(\mathbf{x}_i)$ and $\Phi(\mathbf{x}_j)$ do we need?

- $\approx n^2/2$ dot products
- each dot product costs $\approx d^2/2$ additions and multiplications
- total cost: $\approx n^2 d^2/4$ – quite large!

Can we reduce this cost?

Simplifying the dot product calculation

$$\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b}) = \begin{bmatrix} 1 \\ \sqrt{2}a_1 \\ \sqrt{2}a_2 \\ \dots \\ \sqrt{2}a_d \\ a_1^2 \\ a_2^2 \\ \dots \\ a_d^2 \\ \sqrt{2}a_1a_2 \\ \sqrt{2}a_1a_3 \\ \dots \\ \sqrt{2}a_1a_d \\ \sqrt{2}a_2a_3 \\ \dots \\ \sqrt{2}a_{d-1}a_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \sqrt{2}b_1 \\ \sqrt{2}b_2 \\ \dots \\ \sqrt{2}b_d \\ b_1^2 \\ b_2^2 \\ \dots \\ b_d^2 \\ \sqrt{2}b_1b_2 \\ \sqrt{2}b_1b_3 \\ \dots \\ \sqrt{2}b_1b_d \\ \sqrt{2}b_2b_3 \\ \dots \\ \sqrt{2}b_{d-1}b_d \end{bmatrix} = \left\{ \begin{array}{l} 1 \\ + \\ 2 \sum_{i=1}^d a_i b_i \\ + \\ \sum_{i=1}^d (a_i b_i)^2 \\ + \\ 2 \sum_{i=1}^d \sum_{j=i+1}^d a_i a_j b_i b_j \end{array} \right.$$

Simplifying the dot product calculation

Consider now the following different function of \mathbf{a} and \mathbf{b} :

$$\begin{aligned}
 (\mathbf{a} \cdot \mathbf{b} + 1)^2 &= (\mathbf{a} \cdot \mathbf{b})^2 + 2\mathbf{a} \cdot \mathbf{b} + 1 \\
 &= \left(\sum_{i=1}^d a_i b_i \right)^2 + 2 \sum_{i=1}^d a_i b_i + 1 \\
 &= \sum_{i=1}^d \sum_{j=1}^d a_i b_i a_j b_j + 2 \sum_{i=1}^d a_i b_i + 1 \\
 &= \sum_{i=1}^d (a_i b_i)^2 + 2 \sum_{i=1}^d \sum_{j=i+1}^d a_i b_i a_j b_j + 2 \sum_{i=1}^d a_i b_i + 1
 \end{aligned}$$

This is exactly the definition of $\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$ from the previous slide!

- Explicitly calculating $\Phi(\mathbf{a}) \cdot \Phi(\mathbf{b})$ costs $O(d^2)$.
- Calculating $(\mathbf{a} \cdot \mathbf{b} + 1)^2$ costs only $O(d)$!

Even higher-order polynomials

Raising $(\mathbf{a} \cdot \mathbf{b} + 1)^p$ to any power p still costs $O(d)$.

Comparing costs:

p	Cost(naïve Gram)	$n = 100?$	Cost(smarter Gram)	$n = 100?$
2	$d^2 n^2 / 4$	$2,500n^2$	$dn^2 / 2$	$50n^2$
3	$d^3 n^2 / 12$	$83,000n^2$	$dn^2 / 2$	$50n^2$
4	$d^4 n^2 / 48$	$1,960,000n^2$	$dn^2 / 2$	$50n^2$

'Gram' means the Gram matrix, the matrix of inner products of all pairs of vectors.

- 'naïve Gram' means calculating $\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$ explicitly with the basis expansion
- 'smarter Gram' means calculating $(\mathbf{x}_i \cdot \mathbf{x}_j + 1)^p$

Further benefits of this shortcut

Suppose we have

- data in $d = 100$ dimensions
- a polynomial basis expansion with $p = 5$

Each dot product now needs 103 operations, instead of 75 million.

But are there potential problems remaining?

Problem #1

What about overfitting in 75 million dimensions?

Problem #1

What about overfitting in 75 million dimensions?

The issue goes away due to maximizing the margin!

- there are really only n parameters being fit – the Lagrange multipliers $\alpha_1, \dots, \alpha_n$
 - most are set to zero due to maximizing the margin
- asking for small $\|\mathbf{w}\|$ is like weight decay in neural networks – designed to smooth the learned function and reduce overfitting

Problem #2

Calculating $f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \Phi(\mathbf{x}) - b)$ costs 75 million operations!

Problem #2

Calculating $f(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \Phi(\mathbf{x}) - b)$ costs 75 million operations!
 – **No it doesn't!** Remember that

$$\mathbf{w} = \sum_{i:\alpha_i>0} \alpha_i y_i \Phi(\mathbf{x}_i)$$

The sum is over the support vectors. Therefore:

$$\begin{aligned} \mathbf{w} \cdot \Phi(\mathbf{x}) &= \left(\sum_{i:\alpha_i>0} \alpha_i y_i \Phi(\mathbf{x}_i) \right) \cdot \Phi(\mathbf{x}) \\ &= \sum_{i:\alpha_i>0} \alpha_i y_i (\Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x})) \\ &= \sum_{i:\alpha_i>0} \alpha_i y_i (\mathbf{x}_i \cdot \mathbf{x} + 1)^5 \end{aligned}$$

This costs just $O(ds)$, where $s = |\{i : \alpha_i > 0\}|$ is the number of support vectors.

Kernel functions!

What we've looked at is an example of a (polynomial) kernel function:

$$K(\mathbf{a}, \mathbf{b}) = (\mathbf{a} \cdot \mathbf{b} + 1)^p$$

There are other very high-dimensional basis functions that can be made practical by finding the right kernel:

- RBF-style kernel:

$$K(\mathbf{a}, \mathbf{b}) = \exp\left(-\frac{\|\mathbf{a} - \mathbf{b}\|^2}{2\sigma^2}\right)$$

- Neural-net-style kernel:

$$K(\mathbf{a}, \mathbf{b}) = \tanh(\kappa \mathbf{a} \cdot \mathbf{b} - \delta)$$

σ , κ , and δ are parameters that must be chosen by a model selection method like cross-validation.

Kernels and the 'kernel trick'

Kernels are a substitute for a dot product, where the idea is:

- transform the data \mathbf{x} to some other space $\Phi(\mathbf{x})$, which might be of extremely high dimension
- compute the dot product in the other space

The 'kernel trick' is to do the mapping implicitly by:

- replacing dot products with kernel function evaluations
- never explicitly calculating the $\Phi(\mathbf{x})$ mapping

Benefits:

- huge computational savings
- substituting different kernels gives different learning algorithms