MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2003
Recitation 11, November 13 &14, 2003

Learning: 3 Demos and a Funeral (Perceptrons &Neural net; GAs; SVMs Prof. Robert C. Berwick

Agenda

1. **Perceptrons & Neural Nets: quick backprop review**
2. **Genetic Algorithms**
3. **Support Vector machines: see slides**

1. Neural Networks & Perceptrons: summary equations

a. The Forward Equation:

$$y = f(z) \qquad Threshold(z) = \begin{cases} 1 & if\ z > 0 \\ 0 & otherwise \end{cases}$$

$$Sigmoid(z) = \frac{1}{1+e^{-x}}$$

$$z = \sum_i w_i x_i$$

Output is a function of z, the sum of the weighted inputs. The function *f* can be a threshold or a sigmoid.

b. Threshold learning for a simple perceptron:

$\Delta w = -rx(y - y^*)$ Change in weight = - LearningRate * Input * Error

c. Sigmoid learning for an output node:

A sigmoid function can be differentiated to get the slope dy/dz = y(1-y).
Using this slope to adjust the weights gives:

$\Delta w = -rx(y - y^*)y(1-y)$ Change in weight = - Learning Rate * Input * Error * Slope

e. Sigmoid learning for a hidden (non-output) node:

The difference for hidden nodes is that error is defined in terms of the error of the later nodes:

$\Delta w_{ij} = -rx\delta_j$ Change in weight = - Learning Rate * Input * Delta

For output nodes, delta is the slope times the error.
For hidden nodes, delta is the slope times the sum of the weighted deltas of the next layer of nodes.

$$\delta_j = \begin{cases} y_j(1-y_j)(y_j - y_j^*) & \text{If node i is an output unit} \\ y_j(1-y_j)\sum_k \delta_k w_{jk} & \text{If node i is not an output} \end{cases}$$

2. Mathematical niceties

3. Genetic Algorithms

The simple GA:

1. Start with a random population of *n* individuals (candidate solutions to the problem).
2. Calculate the fitness $f(x)$ of each individual x in the population.
3. Create a new population of *n* individuals as follows:
   a. Pick individuals to be parents according to their fitness.
   b. With probability p_c cross over the parents' genes to form children.
   c. With probability p_m add a mutation to the child's genes.
4. Go to step 2, with the new population; repeat.

Genes are usually represented by binary strings, such as 0110010100

Mutation
A common type of mutation is a single bit-flip:

Input: 0110010100
Output: 0111010100

Crossover
Crossover combines genes from two parents:

Input 1: 011**0010100**
Input 2: **100**1101111
Output: 1000010100

Fitness: Proportional vs. Rank
Proportional fitness assigns an individual a chance of reproducing in direct proportion to its fitness:

$$P_i = \frac{fitness_i}{totalPopulationFitness} = \frac{f(x_i)}{\sum_j f(x_j)} \quad \text{[Proportional Method]}$$

Rank fitness assigns an individual a chance of reproducing in proportion to its rank in the population.
(In a population of N individuals, the best gets a rank N, the worst gets a rank 1).

$$P_i = \frac{rank_i}{totalPopulationRank} = \frac{r(x_i)}{\sum_j r(x_j)} \quad \text{[Rank Method]}$$

Please see the demos, to be posted, on my web site & additional slides on SVMs, Gas

3. Support vector machines - SVMs (see also slides, posted)

- SVMs are another method for classifying observed data, especially good when you don't really know what features are important, or if they are not linearly separable; also superb at conquering **overfitting (two sources of error: classifying points correctly; and generalization error – compare the perceptron, it stops learning after all training points correct, i.e., no classifier error on training)**
- Train system with +, - examples; then give it new samples to test.
- Result of training are **decision boundaries** and **weights;** these decision boundaries can be lines (hyperplanes); or curves, depending on 'distance measure' used – a more sophisticated notion of feature combination (use the **dot product** of feature values, plus **kernel transformation,** see below).
- **Support vectors** are the data points that lie closest to the decision surface. They are vectors drawn from the origin to the location of the data point in *n*-space.
- They are the points most difficult to classify
- They have a direct bearing on the optimum location of the decision surface: by definition, support vectors are the elements of the training set that would change the position of the decision boundary if removed.
- Decision boundary is designed to **maximize the 'width'** between one class of samples and another – the so-called **margin** or **gutter.** This minimizes the risk of of mis-classifying points - we "charge" for mistakes and so avoid over-fitting.
- The Langrangian trick: how to maximize the margin or gutter width while still finding the decision boundary. This is a constrained optimization problem.
- The problem with linear support vector machines – how to handle nonlinear separability by kernel transformations (which kernels and why). The standard kernels are: linear (
- The dot product trick: we can't compute kernel, but we can compute dot product under kernel anyway

Example applications

•Isolated handwritten digit recognition
•Object recognition
•Speaker identification
•Charmed quark detection
•Face detection in images
•Text categorization

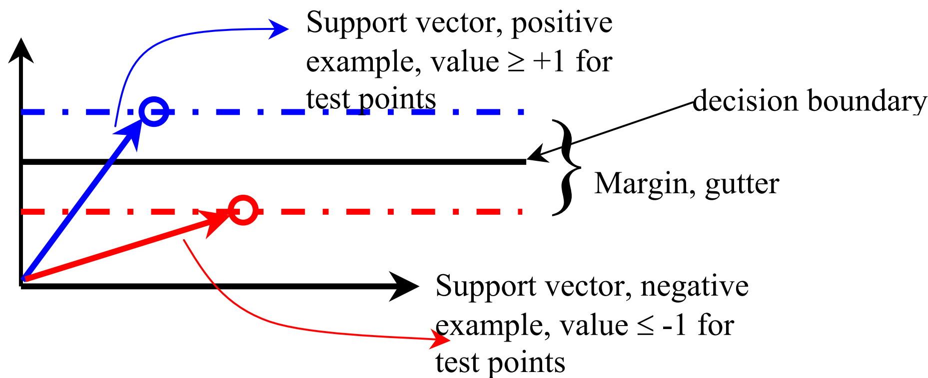
Input to SVM: a set of labeled sample points, measured along some # of features
Output from SVM: A decision boundary and a set of weights. Each sample feature will be given a weight α. If the weight is 0, it doesn't enter into decision boundary. If a feature weight it is positive or negative, it does determine in part where the decision boundary goes and the weight magnitude gives the *relative* contribution of that weight to the optimal decision boundary. The exact shape (line, curve, etc) of the decision boundary will depend on the kernel transform chosen. The actual output from the SVM is the normal, **w,** to the decision boundary, and is defined as:

$$\mathbf{w} = \sum_i \alpha_i y_i x$$

where α are a set of weights, nonzero for the support vectors, a linear combination of the samples. The intuition is that this is now a classifier that **scores** each (new) test data point by weighting it with respect to the (weighted) distance each point is from the non-zero support vectors α. Positive α "drag" the test point towards the positive side of the fence; negative α drag points towards the negative side of the fence.

We shall show that one can find these weights just by using the dot product $x_i x_j$ and this does not depend on x_i in any other way. The search in the weight space will have just one maxima (minima), so we are guaranteed to win (it is a quadratic programming problem). This will also turn out to be the best we can do.

Here is an example to illustrate SVM decision boundaries, margin, and positive and negative



support vectors. We want to maximize the margin, with as few mis-classification errors as possible.

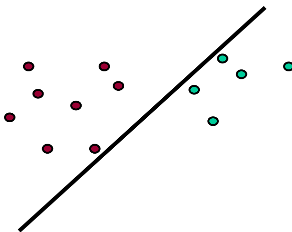
Now the questions we can ask are:

- What shape can the decision boundary be?
- How do we find this shape and position so as to maximize 'street' or margin width?
- How can an SVM fail? (how do we dodge overfitting?)

In a more general (linear) case, the picture looks like this:

Recall that we had a linear example like this: the perceptron. A typical problem: Find *a,b,c,* such that:

$ax + by \geq c$ for red points
$ax + by \leq c$ for green points

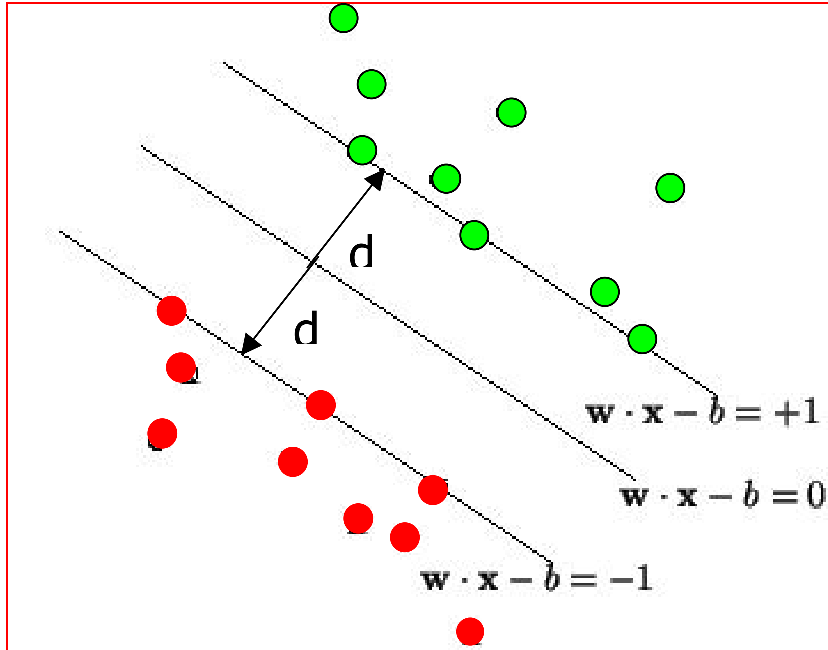


In general, there can be many such *a, b, c* ! How can we find the optimal one? Well, what is *optimal*? We want to use *as few* features as possible (as few parameters – smallest theory, Occam's razor), while making *as few* mistakes as possible (Einstein: a theory should be as simple as possible, but no simpler). SVMs solve this in a principled way – by formulating an optimization problem, rather than greedy search.

Definitions

- Let the set of n training examples (x_i, y_i) where x_i is the feature vector and y_i is the target output (pattern). Let $y_i = +1$ for positive examples and $y = -1$ for negative examples. Assume (for now) that the patterns are linearly separable
- Define the hyperplane H such that:
  $\mathbf{x}_i \cdot \mathbf{w} + b \geq +1$ when $y_i = +1$
  $\mathbf{x}_i \cdot \mathbf{w} + b \leq -1$ when $y_i = -1$

Now we can use the following picture.



H1 and H2 are the planes:

H1: $\mathbf{x_i} \bullet \mathbf{w} + b = +1$

H2: $\mathbf{x_i} \bullet \mathbf{w} + b = -1$

The points on the planes H1 and H2 are the **Support Vectors.** We represent a line via the normal **w** and the distance from the origin, *b.*

d+ = the shortest distance to the closest positive point

d- = the shortest distance to the closest negative point

Note that we are now **penzalizing** bad points (because for red points that are on the **wrong** side of the line, *ax+by-c*

The **margin of** a separating hyperplane is $d^+ + d^-$. = the distance between H1 and H2. We want a classifier with as wide a margin as possible. Let us define this as an optimization problem. Recall the distance from a point(x0,y0) to a line Ax+By+c = 0 is:

$$|A x_0 + B y_0 + c| / \sqrt{(A^2+B^2)}$$

The distance between H and H1 is:

$|\mathbf{w} \bullet \mathbf{x} + b| / ||w|| = 1/||w||$

The distance between H1 and H2 is:

$2/||w||$

So, if we want to **mazimize** the distance between H1 and H2, then we should **minimize** $||w||$ **with the condition that there are no datapoints between H1 and H2,** i.e.,

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

$\mathbf{x_i} \bullet \mathbf{w} + b \leq -1$ when $y_i = -1$ or, combining these two equations:

y ($\mathbf{x_i} \bullet \mathbf{w}$) ³ 1 or $y_i(\mathbf{w} \bullet x_i + b) - 1 = 0$

This is a **constrained** optimization problem, and is well-known to be solvable for this linear case as a quadratic programming problem by the method of **Langrange multipliers.** With this approach, we can show that one can minimize $||w||$ or, equivalently, because it will make the math work out, $\frac{1}{2}||w||^2$, *along with the*

constraint that $y_i(\mathbf{w}\bullet x_i + b) - 1 = 0$. In general, we have

$$L(x,\lambda) = f(x) + \sum_i \alpha_i g_i(x) \text{ a function of } n+m \text{ variables}$$

n for the $x's$, m for the α. Differentiating gives $n+m$ equations, each set to 0.
Why do we do this differentiation of L to solve the problem? As usual – to find a maximum (minimum). Intuitively (see the slides), with the derivative set to 0 we are at the tangent of both the weight function *f* and the constraint curve *g*. At this point, a possible solution point, the normals (perpendiculars) to *f* and *g* are both parallel. The partial derivatives wrt *x* recover the parallel normal constraint; the partial derivatives wrt α recover the *g* constraint

In our case, *f*(*x*): $\frac{1}{2}\|\mathbf{w}\|^2$; *g*(*x*): $y_i(\mathbf{w}.x_i + b) - 1 = 0$ so the Lagrangian is
$L = \frac{1}{2}\|\mathbf{w}\|^2 - \Sigma_i \alpha_i[y_i(\mathbf{w}.x_i + b) - 1]$ where the α_I are the 'Lagrange multipliers'.
Recall that $\mathbf{w} = \Sigma_i \alpha_i y_i x_i$

What do the multipliers alpha mean? Intuitively, each α_I is the rate at which we could increase the Lagrangian if we were to raise the target of that constraint (from zero). But remember that at solution points *p*, $L(p,\lambda) = f(p)$. So the rate of increase of the Lagrangian with respect to that constraint is also the rate of increase of the maximum **constrained** value of *f* with respect to that particular constraint. Note that in our special problem, the alphas also turn out to be the weights – how much each point contributes to pulling apart the margin – so, this makes sense: the alpha is zero if a point does nothing, and is positive or negative corresponding to its 'tug' on the gradient at the decision boundary.

Intuition: think of the constraint $g(x,y)=0$ as a curve in 2-space. We travel along this constraint curve looking for the spot where $f(x, y)$ is maximized (hence derivative is zero).

What is the partial of *L* wrt *w?*

$$\frac{\partial L}{\partial w} = \mathbf{w} - \sum \alpha_i y_i x_i = 0 \text{ or } \mathbf{w} = \sum \alpha_i y_i x_i$$

Plugging this back into the formula for *L* and using the other derivative set to zero, plus some algebra (we won't show this here) yields the revised (dual) optimization problem to *minimize*:

$$\sum \alpha_i - \frac{1}{2}\left(\sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i x_j\right)$$

Note that this all boils down to saying that we want to **find the best α_ι using the dot product between the two sample points $x_i \bullet x_j$. But in this form, all we compute is the dot product, we don't need the training data in any other way. So the dot product is important in finding the support vectors.**

The use of the dot product makes sense as a measure of similarity. Recall that the dot product is the cosine of the angle between two vectors. If two vectors x_1, x_2 are identical, then their dot product is 1; if they are completely different, or orthogonal, then their dot product is 0.

The output from this optimization will be the alpha weights that define the decision boundary. Thus, we will also use the dot product in testing: we use them to figure out which side of the boundary line an unfamiliar point is.

Are we done?

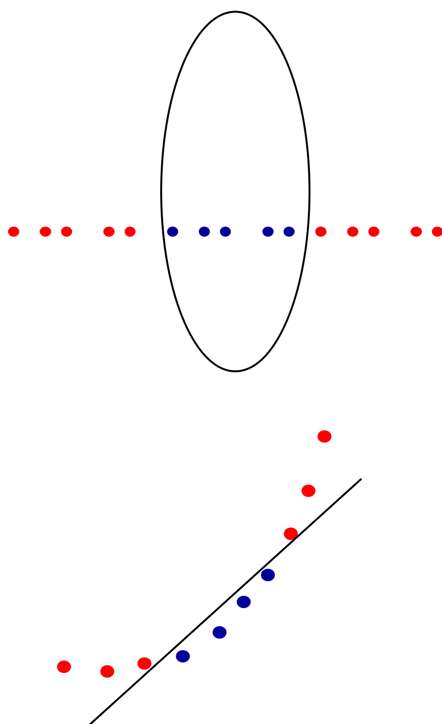
No. Many patterns are not linearly separable! What do we do? We map the original space to a new space, via a function Φ. Then we use a **different** dot product, called a **kernel K** to separate out

the regions linearly in a (higher dimension, transformed) space. Since it is only this dot product we need, we **never** need to compute Φ explicitly!

$$K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \Phi\left(\mathbf{x}_i\right) \cdot \Phi\left(\mathbf{x}_j\right)$$

This doesn't say **how** to find such a K – but we already know some that we can use (we have already used one!) Examples:

Consider the function $x^2 - (a+b)x + ab$. This looks like the following, so it's not linearly separable.



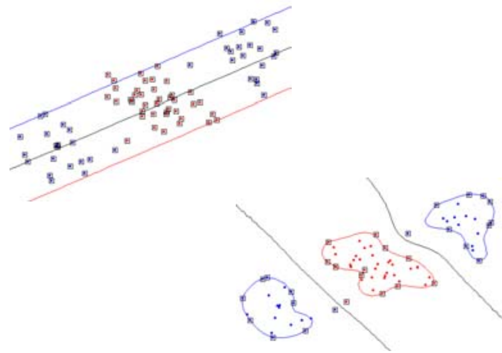
BUT if we apply the transform $x \mapsto \{x^2, x\}$ then this 'straightens out' the curve. Note that the only change to the optimization problem is from this:

$$L = \sum_i \alpha_i - \tfrac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \left(\mathbf{x}_i \bullet \mathbf{x}_j\right)$$

to this:

$$L = \sum_i \alpha_i - \tfrac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K\left(\mathbf{x}_i, \mathbf{x}_j\right)$$

Besides the regular dot product, the the most commonly used transforming dot product is one you've already met before? Can you guess? It is $K(\mathbf{x}, \mathbf{y}) = \tanh(\kappa \mathbf{x} \cdot \mathbf{y} - \delta)$ (i.e., the sigmoid!) The other commonly used dot products are polynomial functions (as above), gaussians, and radial basis functions. Each 'warps' the higher dimensional space so as to linearize it (it is a remarkable result of mathematics that **any** function can be linearly separated in this way – if we map it into a high enough space. We give a picture of what the Gaussian transform 'does':



The (kernel) equation for a Gaussian is $K(\mathbf{x},\mathbf{y}) = \exp\left\{-\|\mathbf{x}-\mathbf{y}\|^2 / 2\sigma^2\right\}$.

Note that we can 'twiddle' the spread of each Gaussian by increasing or decreasing sigma, but this is usually given *a priori.* We can also twiddle the x-y difference, which yields a peak at a different location. Note that superimposed basis functions like these are **radial** basis functions (because when flattened they are concentric circles).

Summarizing: Key ideas here

–Nonlinear mapping of an input vector into a high-dimensional feature space that is hidden from both the output and the input

–Construction of an optimal hyperplane for separating the features discovered in step 1, using Langrangian method

Output is the SVMs, the weights corresponding to the support vectors.