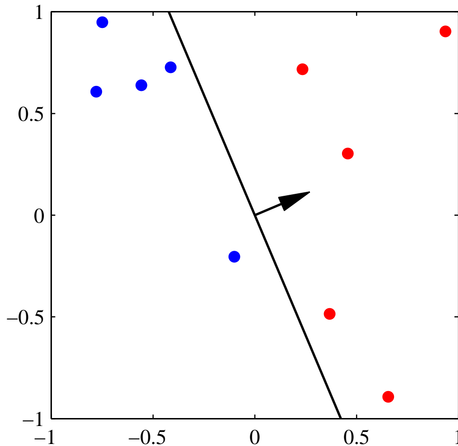
Machine Learning (CS 5350/CS 6350) *9 Sep 2008*

Perceptron

The whole idea behind linear classifiers is to directly represent the *decision* boundary (in binary classification problems). In decision trees and KNN, it is *implicit.*

We begin with the simplest kind of decision boundary: a hyperplane (a flat thing). We put points of each class on either side of the boundary. The mostly-vertical line in the figure below is a possible decision boundary between the blue (left) and red (right) points.



As a *representation* of a decision boundary, we store the vector that is orthogonal (aka "normal") to the decision boundary. We usually call this $\boldsymbol{w}$ (for "weights"... this will become clear soon). For instance, if the x-axis is the first coordinate and the y-axis is the second coordinate, the weight vector in the figure above (the arrow) might have value $\langle 0.4, 0.1 \rangle$, indicating that it is pointing right a lot and up a little.

The representation in terms of $\boldsymbol{w}$ is overkill: if $\boldsymbol{w}$ represents the hyperplane, then $\alpha\boldsymbol{w}$ also does, for any $\alpha > 0$ (it just makes the vector longer, but doesn't change its direction). Thus, we'll usually assume that $||\boldsymbol{w}|| = 1$ (the 1 is for convenience; any positive constant would be fine).

This representation also assumes that the decision boundary passes through the origin. We will shift the hyperplane by adding an addition "bias" term b. In the above figure, if we wished to move the decision boundary in the direction of the arrow, we would use $b < 0$; to move it in the opposite direction, we use $b > 0$.

Given a representation of the decision boundary in terms of $\boldsymbol{w}$ and b, we make our classification according to $\hat{y} = \text{sign}(\boldsymbol{w}^\top \boldsymbol{x} + b)$, where $\boldsymbol{x}$ is an input and $\hat{y}$ is the predicted class (note that we refer to our two classes as $+1$ and -1 instead of 0 and 1).

We can think about the hyperplane as being defined as the set $\{\boldsymbol{x} : \boldsymbol{w}^\top \boldsymbol{x} + b = 0\}$... in other words, it's the set of *all* points for which our classifier is totally uncertain.

In general, if there exists a hyperplane that perfectly separates our data onto two sides, we refer to the data as *linearly separable.* Except in very peculiar circumstances, if data is linearly separable, then there are infinitely many separating hyperplanes (just jiggle the one in the figure a little bit any direction).

We usually want to pick out the "best" hyperplane and the criteria that is often chosen is the *maximal margin hyperplane*, which is essentially the one with the most "wiggle room." Formally, the *margin* of a hyperplane (denoted γ is the distance to the closest point): $\gamma = \min_n y_n(\boldsymbol{w}^\top \boldsymbol{x}_n + b)$.

The next algorithm we discuss is an *error-correction* based algorithm. It maintains a current classifier h and uses this to make predictions. At each step, if h is right, it do nothing. If h is wrong, the update it so that it is "less wrong."

Our representation is linear: we have a weight vector $\boldsymbol{w}$ and a bias b and make predictions on $\boldsymbol{x}$ of the form $\text{sign}[\boldsymbol{w}^\top \boldsymbol{x} + b]$.

The perceptron uses simple additive updates:

1. Initialize $\boldsymbol{w} = 0$, $b = 0$
2. For $n = 1 \dots$
   (a) Obtain $\boldsymbol{x}_n$
   (b) Predict $\hat{y}_n = \text{sign}[\boldsymbol{w}^\top \boldsymbol{x}_n + b]$
   (c) Suffer loss ℓ_n (zero if $\hat{y}_n = y_n$, one otherwise)
   (d) If $\ell_n = 0$, continue on as before. Otherwise, update:

$$\begin{aligned} \boldsymbol{w} &\leftarrow \boldsymbol{w} - \hat{y}_n \boldsymbol{x}_n \\ b &\leftarrow b - \hat{y}_n \end{aligned} \tag{1}$$

The idea with the algorithm is that with binary classification, if $\hat{y}_n$ was wrong, then $-\hat{y}_n$ is *right*. What we do in update (1) is to move $(\boldsymbol{w}, b)$ closer to making the right prediction. Why? Let $(\boldsymbol{w}', b')$ denote the updated classifier. Let's look at what we *would* predict, if we received the same $\boldsymbol{x}$ again. Suppose that the truth is y (so $y = -\hat{y}$); we know we want $y[\boldsymbol{w}'^\top \boldsymbol{x} + b] > 0$.

$$\begin{aligned} y[\boldsymbol{w}'^\top \boldsymbol{x} + b'] &= y(\boldsymbol{w} - \hat{y}\boldsymbol{x})^\top \boldsymbol{x} + y(b - \hat{y}) \\ &= y[\boldsymbol{w}^\top \boldsymbol{x} + b] - y(\hat{y}\boldsymbol{x}^\top \boldsymbol{x} + \hat{y}) \\ &= y[\text{old prediction}] - y\hat{y}\,||\boldsymbol{x}||^2 - y\hat{y} \\ &= y[\text{old prediction}] + ||\boldsymbol{x}||^2 + 1 \end{aligned}$$

In the last step, we used the fact that $y\hat{y} = -1$ if we made an error. So what's happening is we have our old prediction times y (which we know was negative—since we erred) plus $||\boldsymbol{x}||^2 + 1$, which is always positive. Which means that the whole thing is getting more positive. Which is exactly what we want.

This leads directly to the following theorem for *batch* applications:

Theorem (Block & Novikoff): if the training data is linearly separable with margin γ with weights $\boldsymbol{u}$, then Perceptron will terminate in $||\boldsymbol{u}||^2 \le R^2/\gamma^2$ iterations (assuming $||x|| \le R$ for all x).

Proof sketch: (Suppose $R = 1$.) Let $\boldsymbol{w}_k$ be the weights before the kth update, so $\boldsymbol{w}_1 = 0$ and if the kth error is on example n, then $y_n(\boldsymbol{w}_k{}^\top \boldsymbol{x}_n) \le 0$. Then, $\boldsymbol{w}_{k+1}{}^\top u = \boldsymbol{w}_k{}^\top u + y_n(u^\top \boldsymbol{x}_n) \ge \boldsymbol{w}_k{}^\top u + \gamma$, so $\boldsymbol{w}_{k+1}{}^\top u \ge k\gamma$. Also, $||\boldsymbol{w}_{k+1}||^2 = ||\boldsymbol{w}_k||^2 + 2y_n(\boldsymbol{w}_k{}^\top \boldsymbol{x}_n) + ||\boldsymbol{x}_n||^2 \le ||\boldsymbol{w}_k||^2 + 1$ so $||\boldsymbol{w}_{k+1}||^2 \le k$. Thus $\sqrt{k} \ge ||\boldsymbol{w}_{k+1}|| \ge \boldsymbol{w}_{k+1}{}^\top u \ge k\gamma$; then algebra.

For the inseparable case, just force it to be separable (add a feature).

Voting: keep a copy of each value of $\boldsymbol{w}$ and make predictions by voting ($\text{sign}[\sum_i \text{sign}[\boldsymbol{w}_i{}^\top \boldsymbol{x}]]$).

Averaging: use averaged weights to make prediction ($\text{sign}[(\sum_i \boldsymbol{w}_i)^\top \boldsymbol{x}]$).