COS 424: Interacting with Data

Lecturer: Robert Schapire
Scribe: Jia Deng

Lecture #7
Feb 27, 2007

1 Review of boosting

Last time we talked about boosting. The idea is to combine many weak learners to get a more accurate prediction. We rely on the weak learning assumption that the weak learners consistently predict better than random guessing.

Recall the three conditions for successful learning. First we need enough data. We have no control over this in Adaboost. Second, we want low training error. Adaboost reduces training error exponentially as the number of training rounds T increases. Finally we prefer simple prediction rules. In Adaboost the complexity of the prediction rules increases with T. However, Adaboosting still performs well with large T. The reason is that with more training rounds Adaboost increases the *margin*, that is, the confidence in its prediction. Therefore Adaboost tends not to overfit and T is not a critical parameter.

2 SVMs

Boosting was not designed to maximize margin but happens to do so. We do have methods that explicitly maximize margin. One of them is *Support Vector Machine*s, or *SVM*s.

Let's go back to a geometric view as we did with the Nearest Neighbor algorithm. Assume all example points are real number vectors in Euclidean space. For discrete attributes, we can convert them to real numbers, *e.g.* 1 as smoke and 0 as not smoke. The Nearest Neighbor algorithm has trouble with high dimensional data. We will see how SVMs can get around this.

Suppose we have data points as in Figure 1. Our goal is to find a prediction rule that separates the positives and negatives. A natural choice would be to use a line, or a hyperplane in high dimensional space. It is also called a *linear threshold function* (LTF) or *perceptron*.

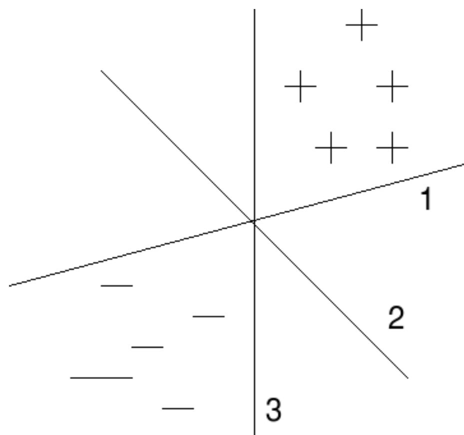

Figure 1: 3 candidate lines

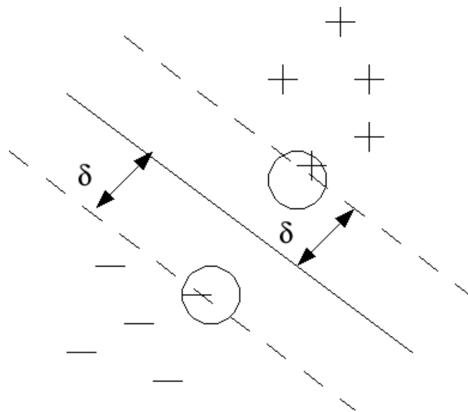

Figure 2: Maximize margin

We have many candidate lines, for example, the three lines in Figure 1. Line(2) seems to be more natural in that it separates all points as much as possible. Line(1) and line(3) are not as good because if we jiggle them a little, they tend to make different predictions for nearby points. In other words, line(1) and line(3) are not as confident when making predictions.

The idea of SVMs algorithm is to maximize the distance to all points. We can imagine that there is a band region with no points in it, as in Figure 2. We want to find a line that maximizes the width of the band. We define the (one side) width of the band, δ, to be the *margin*. Note that the margin defined here is different from that with boosting. Here the margin is defined on the entire training set whereas in boosting it is on individual points. We call the closest points (those with exact distance δ and circled in Figure 2) the *support vectors*.

3 Complexity of SVMs' predictions

Why is this a good idea? Let's go back to the three conditions for successful learning. We have no control over whether we have enough data. The training error is zero. (For now let's assume that the examples are always linearly separable.) What about the complexity of the predictions? Since we have infinite number of hypotheses, we cannot use $\ln |H|$. Instead we use VC-Dimension. It turns out that the VC-Dimension of linear threshold functions in $\mathbb{R}^n$ is n, *i.e.*

$$\text{VC-dim(LTF's in } \mathbb{R}^n) = n \; .$$

(Note that here and throughout this development, we are assuming the hyperplanes pass through the origin.) This is undesirable because the amount of data we need would be roughly proportional to n, which is true for general LTF's. However, for LTF's with margin δ for points in a ball of radius R, we have

$$\text{VC-dim(LTF's with margin } \delta \text{ for points in ball of radius R)} = \left(\frac{R}{\delta}\right)^2 \; .$$

This is nice because the complexity is independent of the dimension n. Also VC-Dimension decreases as the margin δ gets larger.

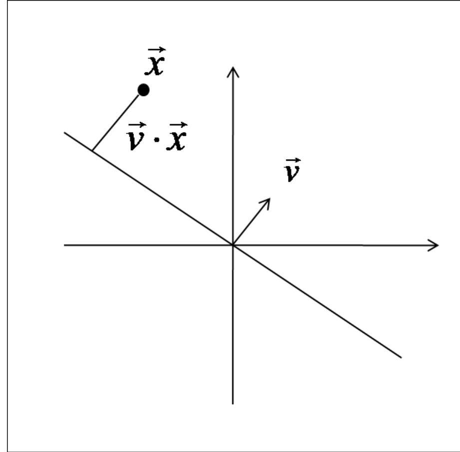

Figure 3: linear algebra review

4 Mini linear algebra review

Now we will have a tiny quick review of linear algebra. For simplicity, we are only considering lines (hyperplanes) that go through the origin. Any hyperplane can be defined by its unit normal vector $\boldsymbol{v}$, that is,

$$\text{hyperplane} = \{\boldsymbol{x} \mid \boldsymbol{v} \cdot \boldsymbol{x} = 0\} \ .$$

For an arbitrary point $\boldsymbol{x}$, the (signed) distance to the hyperplane is $\boldsymbol{v} \cdot \boldsymbol{x}$. To summarize,

$$\boldsymbol{v} \cdot \boldsymbol{x} = \begin{cases} = 0 & \text{if } \boldsymbol{x} \text{ is on the hyperplane} \\ > 0 & \text{if } \boldsymbol{x} \text{ is above the hyperplane} \\ < 0 & \text{if } \boldsymbol{x} \text{ is below the hyperplane} \end{cases} \ .$$

5 Computing the classifier

Now let's look at how to compute the maximum margin hyperplane. Given $(\boldsymbol{x_1}, y_1), \ldots, (\boldsymbol{x_m}, y_m)$, where $\boldsymbol{x_i} \in \mathbb{R}^n$ are data points and $y_i \in \{-1, +1\}$ are labels, we want to find a hyperplane $\boldsymbol{v}$ maximizing the margin δ, that is,

$$\boxed{\begin{array}{l} \text{Maximize } \delta \\ \qquad \text{Subject to} \begin{cases} \boldsymbol{v} \cdot \boldsymbol{x_i} \geq \delta & \text{if } y_i = +1 \\ \boldsymbol{v} \cdot \boldsymbol{x_i} \leq -\delta & \text{if } y_i = -1 \\ \|\boldsymbol{v}\| = 1 \end{cases} \end{array}}$$

We can rewrite the constraints more compactly as

$$\begin{cases} y_i(\boldsymbol{v} \cdot \boldsymbol{x_i}) \geq \delta, \ i = 1, \ldots, m \\ \|\boldsymbol{v}\| = 1 \end{cases} \ .$$

We introduce a new variable $\boldsymbol{w}$:

$$\boldsymbol{w} \triangleq \frac{\boldsymbol{v}}{\delta} \ .$$

Substituting $\boldsymbol{v}$ with $\delta \boldsymbol{w}$ in the constraints we get

$$\begin{cases} y_i(\boldsymbol{w} \cdot \boldsymbol{x_i}) \geq 1, \ i = 1, \ldots, m \\ \|\boldsymbol{w}\| = \frac{1}{\delta} \end{cases} \ .$$

Observe that maximizing δ is equivalent to minimizing $\|\boldsymbol{w}\|$, or $\frac{1}{2}\|\boldsymbol{w}\|^2$, since $\|\boldsymbol{w}\| = \frac{1}{\delta}$. Therefore, we can eliminate the second constraint and reformulate the problem as

Minimize $\frac{1}{2}\|\boldsymbol{w}\|^2$

Subject to $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i}) \geq 1,\ i = 1, \ldots, m$

This is a convex programming problem. For reasons we will see later, we want to convert it to its *dual problem*. Following a standard recipe, first we write down the *lagrange function*

$$L(\boldsymbol{w}, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\| - \sum_i \alpha_i \left(y_i \left(\boldsymbol{w} \cdot \boldsymbol{x_i}\right) - 1\right)$$

where α_i is the *lagrange multiplier*. Then we take the partial derivatives of L with respect to $\boldsymbol{w}$. Solving equations

$$\frac{\partial L}{\partial w_j} = 0,\ j = 1, \ldots, n$$

we get

$$\boldsymbol{w} = \sum_i \alpha_i y_i \boldsymbol{x_i}\ . \tag{1}$$

Plugging $\boldsymbol{w}$ into L, we have

$$\bar{L}(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x_i} \cdot \boldsymbol{x_j}\ . \tag{2}$$

By optimization theory, the dual problem is

Maximize $\bar{L}(\boldsymbol{\alpha}) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x_i} \cdot \boldsymbol{x_j}$

Subject to $\alpha_i \geq 0,\ i = 1, \ldots, m$

We can solve this dual problem and use the dual solution $\boldsymbol{\alpha}^*$ to obtain the primal solution $\boldsymbol{w}^*$ by equation(1). There are many efficient ways of solving the dual problem. Anyway, since the objective function is convex, we can follow a path upward without worrying about local maxima. How to solve this problem exactly is beyond the scope of this class.

Observe two properties of the dual problem. First, $\boldsymbol{w}$ is a linear combination of the data points. Second, we only need to compute $\boldsymbol{x_i} \cdot \boldsymbol{x_j}$ for $\bar{L}$, which is very important as we will see pretty soon.

6 Dealing with linearly inseparable data

What do we do if the data are not linear separable? There are two answers.

6.1 Soft margin

One way is to allow the algorithm to move points. Suppose the data are almost linearly separable, with only a few bad examples, as in Figure 4. We give the algorithm permission to move bad points to the other side of the hyperplane. We change the optimization function by adding a new term penalizing large distances. The optimization function becomes

$$\text{Minimize } \frac{1}{2}\|\boldsymbol{w}\|^2 + C \cdot (\text{distance points moved})\ .$$

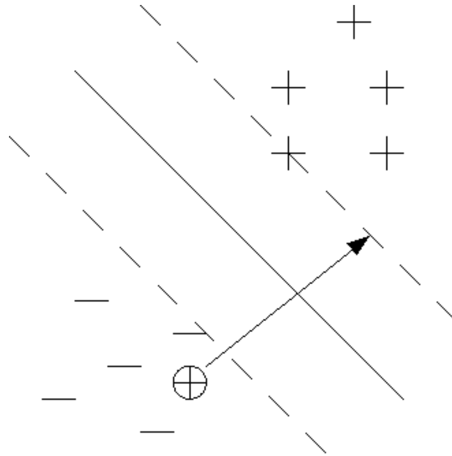

Figure 4: Soft margin

where C is a constant determining how far we allow the points to be moved. This idea is called *soft margin*. In the original proposal, the penalty term was raised to certain power. However, then the problem becomes non-convex, thus more difficult to solve.

6.2 Kernel trick

But what if we have data points like Figure 5 ? It is impossible to separate them using a line. The solution is to map the data into higher dimensional space, where they often become linearly separable.

For example, below is a mapping from 2 dimension to 6 dimension.

$$F(\boldsymbol{x}) = F((x_1, x_2)) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2)$$

A point $(2, 3)$, for example, is mapped to $(1, 2, 3, 6, 4, 9)$. Under this mapping, a hyperplane in 6-D space becomes

$$a + bx_1 + cx_2 + dx_1x_2 + ex_1^2 + fx_2^2 = 0$$

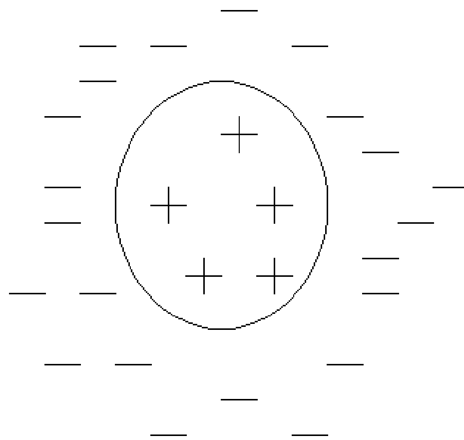

Figure 5: Kernel trick

which is a conic section (ellipse, parabola, *etc.*) in 2-D space. Therefore the data points in Figure 5 can be separated by an ellipse under this mapping.

We can follow the idea of F and add more terms of higher degree to map the data into even higher dimensions. However, if the original dimension is n and we add terms of all degrees up to k , the new dimension is $O(n^k)$,*i.e.* exponentially increasing, which seems a very bad situation. First, statistically we need much more data because of the curse of dimensionality. Second, computationally it is much more expensive.

However, recall that

$$\text{VC-dim(LTF's with margin } \delta \text{ for points in ball of radius R)} = \left(\frac{R}{\delta}\right)^2 .$$

It follows that the complexity of the predictions is independent of n. Also note that when mapped into higher dimensions, δ increases so the VC-Dimension actually decreases. In fact, R also gets larger, but often δ gets larger faster. This resolves our first concern.

It also turns out that actually we can avoid explicit computation in high dimensions by using the *kernel trick*. Recall that we only need to compute $\boldsymbol{x_i} \cdot \boldsymbol{x_j}$ for $\bar{L}$. Now we modify the mapping function F a little bit:

$$F(\boldsymbol{x}) = F((x_1, x_2)) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2) .$$

We then compute the inner product of point $\boldsymbol{x}$ and $\boldsymbol{z}$ in high dimensional space, that is,

$$\begin{aligned} F(\boldsymbol{x}) \cdot F(\boldsymbol{z}) &= 1 + 2x_1z_1 + 2x_2z_2 + 2x_1x_2z_1z_2 + x_1^2z_1^2 + x_2^2z_2^2 \\ &= (1 + x_1z_1 + x_2z_2)^2 \\ &= (1 + \boldsymbol{x} \cdot \boldsymbol{z})^2 . \end{aligned}$$

Note that we don't even have to map! We call $F(\boldsymbol{x}) \cdot F(\boldsymbol{z})$ the *kernel* function. Generally, if we map the points using up to degree k, the kernel function is

$$K(\boldsymbol{x}, \boldsymbol{z}) = (1 + \boldsymbol{x} \cdot \boldsymbol{z})^k .$$

There are of course lots of other kernels, for example

$$K(\boldsymbol{x}, \boldsymbol{z}) = \exp(-c\|\boldsymbol{x} - \boldsymbol{z}\|^2) .$$

Kernels measure similarity between points. Figuring out what are good kernels for a particular application is a big industry.

So far we can get around the computational burden in training. What about testing? We replace $\boldsymbol{x_i}$ in equation(1) by $F(\boldsymbol{x_i})$, that is,

$$\boldsymbol{w} = \sum_i \alpha_i y_i F(\boldsymbol{x_i}) .$$

When evaluating a test point $\boldsymbol{z}$, we compute

$$\begin{aligned} \boldsymbol{w} \cdot F(\boldsymbol{z}) &= \sum_i \alpha_i y_i F(\boldsymbol{x_i}) \cdot F(\boldsymbol{z}) \\ &= \sum_i \alpha_i y_i K(\boldsymbol{x_i}, \boldsymbol{z}) . \end{aligned}$$

Next time we will talk about clustering and probabilistic methods.