CS-621 Theory Gems September 27, 2012

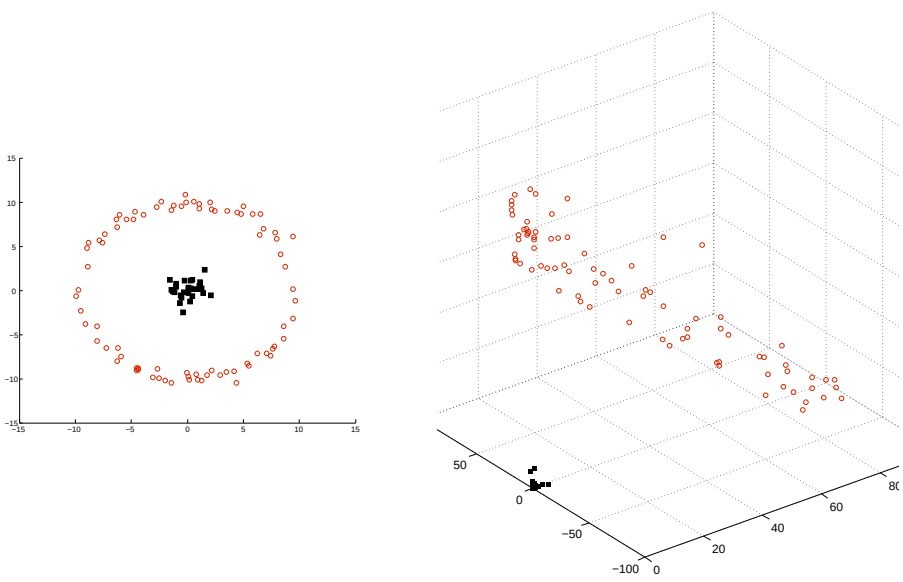
Lecture 4

Lecturer: Aleksander Mądry *Scribes: Alhussein Fawzi*

1 Learning Non-Linear Classifiers

In the previous lectures, we have focused on finding linear classifiers, i.e., ones in which the decision boundary is a hyperplane. However, in many scenarios the data points cannot be really classified in this manner, as there simply might be no hyperplane that separates most of the positive examples from the negative ones - see, e.g., Figure 1 (a).

Clearly, in such situations one needs to resort to more complex (non-linear) classifiers and thus one would expect that there is no use here for the linear classification algorithms we developed so far. Fortunately, as we will see in this lecture, this is not really the case as there actually are powerful and convenient ways of performing a non-linear classification by building on the algorithms for the linear one. In particular, we will see two very useful and quite broadly-applicable techniques: the *Kernel Trick* (or just *kernelization*) and *boosting*.



(a) Original data samples (b) Samples in the high dimensional space

Figure 1: Mapping data samples to a high dimensional space can make them linearly separable.

2 The Kernel Trick

To describe the Kernel Trick, let us consider the example depicted in Figure 1. In this simple case, we have data points living in $\mathbb{R}^2$ and it is easy to see that there is no good linear classifier here.

However, the key observation to make is that if we appropriately map this two-dimensional data into a higher dimension (namely, $\mathbb{R}^3$),then linear separation (in this new host space) becomes possible. More

precisely, consider the mapping $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ given by

$$\phi((x_1, x_2)) \to (z_1, z_2, z_3) = (x_1^2, \sqrt{2}x_1x_2, x_2^2).$$

The result of applying this mapping to our data from Figure 1 (a) can be seen in Figure 1 (b). Clearly, it is now possible to separate the samples with a three-dimensional hyperplane described by some vector $w = (w_1, w_2, w_3)$ and an offset θ. Furthermore, we can map this hyperplane back from $\mathbb{R}^3$ to $\mathbb{R}^2$. To do that, note that the points (z_1, z_2, z_3) belonging to this hyperplane in $\mathbb{R}^3$ must satisfy $w \cdot z = \theta$. By inverting our mapping ϕ, we get that a point $x = (x_1, x_2)$ belongs to the projection of this hyperplane back in $\mathbb{R}^2$ iff it satisfies the following equation

$$w_1x_1^2 + w_2\sqrt{2}x_1x_2 + w_3x_2^2 = \theta.$$

It is not hard to see that this equation describes an ellipse in $\mathbb{R}^2$. So, in this way, we managed to use linear classification algorithm to find a (highly) non-linear classifier.

The above example provides a very appealing approach to dealing with a data that is not linearly separable. However, before this technique can be really useful in general setting, it has some important shortcoming that we need to address. Namely, even if we know a mapping ϕ whose application will make the data linearly separable, applying it to our dataset might be computationally quite expensive. Also, running the linear classification algorithm on the (very) high-dimensional image of our data might add considerable running time overhead.

Fortunately, these complications can be easily avoided. Indeed, note that the linear classification algorithms that we have studied so far (Perceptron, Winnow algorithm, and SVMs) interact with the data in a very specific way – all they need to do is to be able to compute inner products of the (mapped) data points. More precisely, the only information that these algorithms require is to be able to compute an inner product of a given sample x^j and the candidate classifier w - based on this information alone these algorithms can compute the final classifier w_f. Furthermore, all these algorithms have a key property that each candidate classifier they use (as well as, the final one) can be always represented as a combination of some data points. So, imagine that for each i and j, one was supplied with the value $K(x^i, x^j)$ of the inner product of the *mappings* of vectors x^i and x^j, i.e., $K(x^i, x^j) = \phi(x^i) \cdot_H \phi(x^j)$ (where $\cdot_H$ denotes inner product in the high-dimensional space we perform linear separation in). Clearly, one could compute the inner product needed by the algorithm as:

$$w \cdot_H \phi(x^i) = (\sum_j c_j \phi(x^j)) \cdot_H \phi(x^i) = \sum_j c_j K(x^i, x^j),$$

where c_js are coefficients of the combination of data points that describes w.

To make it more concrete, below is the "kernelized" version of Perceptron algorithm:

1. Start with $c^t \leftarrow (0, \cdots, 0) (\in \mathbb{R}^m)$
2. In round t:
   - Check if $\exists j_t$ with $\sum_{i=1}^{m} c_i^t K(x^i, x^{j_t}) \leq 0$.
   - If not: output c^t.
   - Otherwise: set $c^{t+1} \leftarrow c^t + \delta^{j_t}$, where $\delta_i^{j_t} = 1$ if $i = j_t$ and 0 otherwise.

Note that we have made here a slight modification of the "original" Perceptron algorithm (as seen in Lecture 2). Indeed, as opposed to learning the normal vector of the separating hyperplane w (which might be of very large dimension), we maintain only its implicit representation via the coefficients associated with each data point. (That is, we have $w^t = \sum_{i=1}^{m} c_i^t \phi(x^i)$, for each t.)

Now, all of the above means that there is no need to ever compute (or explicitly work with) the high-dimension mapping of our data. All one needs to know is just the *kernel* of this mapping, i.e., $K(\cdot,\cdot)$!
Note that in our example this kernel is

$$K((x_1,x_2),(y_1,y_2)) = x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2 = ((x_1,x_2)\cdot(y_1,y_2))^2,$$

so it can be easily computed using the inner product of the original (lower-dimensional) space.

In general, a popular choice of kernels (that we used also in our example here) are so-called *polynomial kernels*:

$$\begin{aligned} K_1(x,y) &= (1+x_1y_1)(1+x_2y_2)\cdots(1+x_ny_n) \\ K_2(x,y) &= (1+x\cdot y)^d \end{aligned}$$

These kernels map the vectors into a space of products of their subsets, i.e.,

$$\phi((x_1,\ldots,x_n)) = (1,x_1,\ldots,x_n,x_1x_2,x_1x_3,\ldots).$$

Thus we see that using the Kernel Trick with these kernels not only avoids the explicit computation of the mapping, but also does not require computing directly the scalar products in the corresponding (very) high dimensional space. It is worth pointing out here that using the full polynomial kernel (such as the kernel $K_1(\cdot,\cdot)$ and $K_2(\cdot,\cdot)$ for large value of d) might lead to poor generalization. This is so as the resulting separators are too powerful and can "overfit" the data. Therefore, one usually tends to use polynomial kernels with relatively small degree d.

The general rule of thumb for choosing a kernel is to have it aggregate somehow the inherent similarity of data points in a given problem. For instance, one can see in our example that one reason for the original data to be not linearly separable was that any two data points x^i and x^j such that $x^i = -x^j$, had their inner product negative, even though they are in the same class (as their distance from the origin is the same). On the other hand, after applying our kernel, one would have $K(x^i,x^j) = (x^i\cdot x^j)^2 = ||x^i||^4$ which is positive (as one wants it to be).

In some sense, an "ideal" kernel would be such that (1) $K(x^i,x^j) = 1$ if the labels of x^i and x^j agree, -1 otherwise; and (2) simultaneously have $K(\cdot,\cdot)$ correspond to some simple (and thus well-generalizing) classifier family. (Note that getting a kernel with property (1) alone is very easy - one can just use (1) as a definition of the kernel - the difficulty would be in ensuring that the property (2) holds too.)

3 Boosting

Boosting is a powerful and very general technique that allows one to combine several simple classifiers (that have only mildly good performance on our data) to produce a classifier whose performance is significantly better. To make this precise, we need to introduce two definitions. In what follows, let $\{(x^1,l^1),\ldots,(x^m,l^m)\}$ be our set of samples together with their correct labels.

Definition 1 *A classifier h is* ν-strong, *for some $\nu \geq 0$, if we have $h(x^j) = l^j$ for at least $(1-\nu)$ fraction of samples x^j.*

Roughly speaking, ν-strong classifier, for some sufficiently small ν, is exactly what we would like to obtain.

Definition 2 *A family of classifiers H is* γ-weak, *for some $\gamma > 0$, if for any given distribution $\mathcal{D}$ on the samples, there exists a classifier such that*

$$Pr_{x^j\leftarrow\mathcal{D}}(h(x^j) = l^j) \geq \frac{1}{2} + \gamma.$$

Note that for any distribution $\mathcal{D}$, one can always get a classifier with the classification success of at least $\frac{1}{2}$ by just performing random guessing (or always outputting the label that is more probably under distribution $\mathcal{D}$). Thus, in the definition of the γ-weak classifiers, we want to always be able to get an advantage of at least γ over such a random guess. Also, note that unlike the definition of ν-strong classifier where all the points have the same weight (i.e., one can think that $\mathcal{D}$ is an uniform distribution there), we allow $\mathcal{D}$ to be *any* distribution here.

Now, the boosting technique provides us with a way of obtaining a ν-strong classifier – for as small (but non-zero) ν as we desire – by taking a combination (majority) of just a small number of γ-weak classifiers.

Theorem 1 *For any $\nu > 0$, and H being γ-weak, for some $\gamma > 0$, there exists h_f that is ν-strong and h_f is a combination (majority) of at most $\frac{4}{\gamma^2} \ln\left(\frac{1}{\nu}\right)$ classifiers from H.*

Note that the statement of the theorem does not say anything about efficiently finding such a ν-strong classifier. However, as we will see, the proof of this theorem actually gives us an efficient algorithm for this task, as long as, finding a good-enough classifier in H, for a given distribution $\mathcal{D}$, can be done efficiently.

The proof of the theorem will be based on the Multiplicative Weights Update algorithm and, in the course of it, we will need certain refinement of the performance guarantee of MWU algorithm (cf. the original performance bound given by Theorem 6 in notes from Lecture 1).

Theorem 2 *For any non-empty subset S of n experts, the Multiplicative Weights Update algorithm suffers a total loss l_{MWU} of at most*

$$l_{MWU} \leq \max_{i \in S}\left(\sum_t l_i^t + \varepsilon \sum_t |l_i^t|\right) + \frac{\rho \ln \frac{n}{|S|}}{\varepsilon},$$

where $0 \leq \varepsilon \leq \frac{1}{2}$ and each $l_i^t \in [-\rho, \rho]$.

Intuitively, the above theorem shows that if a significant fraction of the experts (instead of just the best one) has good performance, the loss of MWU algorithm converges to these performance much faster. Proving this theorem is a part of the problem set.

Proof (of Theorem 1)

As already mentioned, to prove this theorem, we will use the Multiplicative Weights Update algorithm. The setup of the learning-from-expert-advice framework is as follows: we have m "experts" (one expert per sample), and we set $\epsilon = \gamma$ and $\rho = 1$. In each round t, we do the following:

1. Let $(p_1^t, \ldots, p_m^t)$ be the convex combination provided by the MWU algorithm.
2. Find $h^t \in H$ such that $\Pr_{x^j \leftarrow p^t}(h^t(x^j) = l^j) \geq \frac{1}{2} + \gamma$ (note that such a h^t exists since H is assumed to be γ-weak).
3. Set the loss of each expert j to be: $l_j^t = 1$ if $h^t(x^j) = l^j$ (correct classification) and 0 otherwise (misclassified sample).

We stop the above procedure after $T = \frac{4}{\gamma^2} \ln\left(\frac{1}{\nu}\right)$ iterations and take $h_f = \text{maj}(h^1, \ldots, h^T)$, where $\text{maj}(\cdot)$ is the majority vote, i.e., $h_f(x) = 1$ if at least half of the classifications $h^1(x), \ldots, h^T(x)$ is 1, and -1 otherwise.

Note that in the above procedure we penalize the experts (samples) that were predicted correctly. This is to ensure that the distribution $(p_1^t, \cdots, p_m^t)$ shifts its focus towards experts (samples) that our classifiers have done pretty poor job on so far. This way we ensure that the subsequent classifiers will focus on these troublesome samples.

Now, to prove that h_f is ν-strong, assume that it is not the case. This means that there exists a subset $\bar{S}$ of cardinality larger than νm such that all samples in this subset are misclassified by h_f.

Since h_f is obtained by a majority vote on the h^ts, any sample in $\bar{S}$ is misclassified by at least $\frac{T}{2}$ h^ts, which in turn means that it is correctly classified by at most $\frac{T}{2}$ of them. Therefore, by construction of our loss functions, the experts corresponding to these samples have relatively small loss. Namely, we have for any $j \in \bar{S}$,

$$\sum_t l_j^t \leq \frac{T}{2}.$$

On the other hand, the total loss of the MWU is given by:

$$\begin{aligned} l_{MWU} = \sum_t l_{MWU}^t = \sum_t \sum_j p_j^t l_j^t = \sum_t \sum_j p_j^t \ \mathrm{I}[h^t(x^j) = l^j] = \sum_t \Pr_{x^j \leftarrow p^t}(h^t(x^j) = l^j) \\ \geq \sum_t (1/2 + \gamma) \\ = T(1/2 + \gamma), \end{aligned}$$

where we used the fact that H is γ-weak and the expression $\mathrm{I}[h^t(x^j) = l^j]$ is 1 if $h^t(x^j) = l^j$ and 0 otherwise.

Now, by making use of Theorem 2 (with $S := \bar{S}$), we finally get:

$$T(1/2 + \gamma) \leq (1 + \gamma)\frac{T}{2} + \frac{\ln\left(\frac{1}{\nu}\right)}{\gamma},$$

where we use the fact that if each expert in $\bar{S}$ has a loss of at most $T/2$ then so has the maximum-loss one.
Simplifying the previous expression, we get $T \leq \frac{2 \ln \frac{1}{\nu}}{\gamma^2}$ which contradicts our choice of $T = \frac{4 \ln \frac{1}{\nu}}{\gamma^2}$. The theorem follows. ■