Introduction to Computational Learning Theory **Spring 2021**

Lecture 5: Linear Classification

Lecturer: Roi Livni *Scribe:*

We so far focused on the setting of *statistical learning theory*. We analysed learning problems in terms of the *sample complexity* and also the *expressive power* of the classes to be learnt. We concluded with analysing Neural Networks and showed how they have theoretical appealing sample complexity as well as expressive power.

We next consider the computational aspects of learning, and we will discuss several learning algorithms. We begin with discussing the perceptron which is a classical algorithm for learning half–spaces, as well as several hardness results. We will then discuss Boosting techniques, which is a useful technique to turn *weak learner* or "finger rules" into a strong prediction rule.

5.1 Efficient Learning of Half–Spaces, Realizable setting

In this section we discuss efficient algorithms to learn halfspaces over the hypercube. we will for simplicity discuss halfspaces without bias i.e.

$$\mathcal{C}_n = \{f_{\mathbf{w}} : f_{\mathbf{w}}(\mathbf{x}) = \mathrm{sgn}(\mathbf{w} \cdot \mathbf{x}), \mathbf{x} \in \{-1, 1\}^n\}.$$

We've already seen that the VC dimension of halfspaces is bounded by $n + 1$[1]. Therefore, if we implement an ERM algorithm over $O(\frac{n}{\epsilon} \log 1/\delta)$ examples, we can obtain an efficient algorithm to learn halfspaces The idea is to implement an ERM through Linear Program (LP). An LP is a problem of the form

[1] In fact, without bias the VC dimension is n

$$\text{minimize } \mathbf{w} \cdot \mathbf{u}$$

$$\text{subject to } A\mathbf{w} \geq \mathbf{v}$$

For some parameters $\mathbf{u}, \mathbf{v}$ and matrix A. Efficient algorithms to solve LP are known to exist (efficient in size of A and $\mathbf{u}$). The first such algorithm was presented in [**?**] and several improvement were constructed since.

Therefore we only need to show that we can formulate the ERM as an LP. To see that we can, first note that if there is a solution, then by pertubation we can assume there is $\mathbf{w}$ such that for all $\mathbf{x}_i \in S$ we have $y_i \mathbf{w} \cdot \mathbf{x}_i > 0$ (i.e. strict inequality). By setting $\gamma = \min y_i \mathbf{w} \cdot \mathbf{x}_i$ and setting $\hat{\mathbf{w}} = \frac{\mathbf{w}}{\gamma}$ we can see that there exists a solution $y_i \hat{\mathbf{w}} \mathbf{x}_i > 1$.

We infact, don't even need $\mathbf{u}$ and we only need to check the feasibility of the problem. Hence, we can now set $\mathbf{u} = 0$ (or any other vector) and set the matrix A such that the i-th row $A_i = y_i \mathbf{x}_i^\top$, and $\mathbf{v} = 1$. By the last paragraph, there is a feasible solution – therefore we can find it using an efficient LP algorithm. Any solution to the above program will satisfy $\text{sgn}(\mathbf{w} \cdot \mathbf{x}_i) = y_i$.

5.1.1 The Perceptron

We next discuss a slightly simpler algorithm that learns halfspaces efficiently (again, in the realizable setting), but with dependence on what we will call *the margin*.

Algorithm 1 Perceptron

Input: A sample $S = \{(x_i, y_i)\}_{i=1}^m$.
Intialize: $\mathbf{w}_0 = 0$
For i=1 to m
Set $\hat{y}_i = \text{sgn}(\mathbf{w}_{i-1} \cdot \mathbf{x}_i)$
Update

$$\mathbf{w}_i \to \mathbf{w}_{i-1} + \frac{1}{2}(y_i - \hat{y}_i)\mathbf{x}_i \qquad \text{i.e.\%} \mathbf{w}_i = \begin{cases} \mathbf{w}_{i-1} & y_i = \hat{y}_i \\ \mathbf{w}_{i-1} - \hat{y}_i \mathbf{x}_i & y_i \neq \hat{y}_i \end{cases}$$

End For

Theorem 5.1. *Let $S = \{(\mathbf{x}_i, y_i)\}_{i=1}^m$ s.t. $\|\mathbf{x}_i\| = R$. Assume S can be separated by a margin γ: Namely,*

there exists $\mathbf{w}^* \in \mathbb{R}^d$ *such that* $\|\mathbf{w}^*\| = 1$ *and*

$$y \cdot \mathbf{w}^* \cdot \mathbf{x} > \gamma$$

for all $(\mathbf{x}, y) \sim D$*. Then the Perceptron algorithm, makes at most* $O(\frac{R^2}{\gamma^2})$ *mistakes. In detail, except for maybe* $O(\frac{R^2}{\gamma^2})$ *iterations, we have that* $\hat{y_i} = y_i$*.*

As an exercise, construct an ERM algorithm for half spaces using the Perceptron algorithm (under the so-called margin assumption).

Proof. Note that whenever the algorithm labels a point $\mathbf{x}_i$ correctly, there is no update. Now let us denote by $\mathbf{w}^{(1)}, \mathbf{w}^{(2)} \ldots$ the sequence of vectors we obtain at each iteration where there was a mistake (i.e. $\mathbf{w}^{(1)}$ is the vector $\mathbf{w}_{t_1}$ where t_1 is the first iteration we see a mistake, $\mathbf{w}^{(2)}$ is $\mathbf{w}_{t_2}$ where t_2 is the second iteration where we see a mistake etc..).

Next, assume the k'th error is made on example t, set $\mathbf{w}^{(k)} = \mathbf{w}_t$, then we have

$$\begin{aligned}
\mathbf{w}^{(k)} \cdot \mathbf{w}^* &= (\mathbf{w}_{t-1} + y_t \mathbf{x}_t) \cdot \mathbf{w}^* \\
&= (\mathbf{w}^{(k-1)} + y_t \mathbf{x}_t) \cdot \mathbf{w}^* \\
&\geq \mathbf{w}^{(k-1)} \cdot \mathbf{w}^* + \gamma
\end{aligned}$$

Where the first equality is from the update rule, the second equality is because $\mathbf{w}_t$ is not updated when there are no mistake. The last inequality follows from the assumption $y\mathbf{w}^*\mathbf{x} \geq \gamma$. It follows by induction on k that $\mathbf{w}^{(k)} \cdot \mathbf{w}^* \geq k\gamma$. By Cauchy Schwartz, we obtain

$$\|\mathbf{w}^{(k)}\| \geq k\gamma. \tag{5.1}$$

Next, recall that we defined $\mathbf{w}^{(k)}$ to the be $\mathbf{w}_t$ where t is the iteration where we see the k-th error. In

particular, $y_t \mathbf{x}_t \mathbf{w}^{(k)} \leq 0$. we have:

$$\begin{aligned}
\|\mathbf{w}^{(k)}\|^2 = \|\mathbf{w}^{(k-1)} + y_t \mathbf{x}_t\|^2 &= \|\mathbf{w}^{(k-1)}\|^2 + \|\mathbf{x}_t\|^2 + 2 y_t \mathbf{x}_t \mathbf{w}^{(k)} \\
&\leq \|\mathbf{w}^{(k-1)}\|^2 + \|\mathbf{x}_t\|^2 && y_t \mathbf{x}_t \cdot \mathbf{w}^{(k)} \leq 0 \\
&\leq \|\mathbf{w}^{(k-1)}\|^2 + R^2
\end{aligned}$$

Hence, again by induction we obtain that

$$\|\mathbf{w}^{(k)}\|^2 \leq k R^2. \tag{5.2}$$

Taking eqs. (5.1) and (5.2) together we obtain that

$$k\gamma \leq \sqrt{k} R.$$

Alternatively

$$k \leq \frac{R^2}{\gamma^2}.$$

□

5.2 Non-Linear Problems

Suppose instead of learning a linear decision boundary, we want to learn some polynomial-threshold function. Specifically, given x we want to learn a $q-$degree polynomial p such that

$$y = \operatorname{sgn}[p(\mathbf{x})] = \operatorname{sgn}\left[\sum_{J \in [\binom{d}{q}]} \mathbf{w}_J \prod_{i \in J} \mathbf{x}_i \right].$$

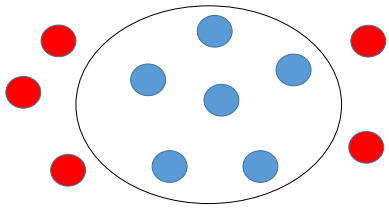

Figure 5.1: An example of non-linear decision boundary of the form $y = \text{sgn}(x_1^2 + x_2^2)$

A first simple solution is a *linear-reduction*. Let $\Phi(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^{\sum_{k \le q} \binom{d}{k}}$ be an embedding to a linear vector space, indexed by all possible multi-indices (i.e. set of indexes) of size less that q (i.e. each coordinate in $\mathbb{R}^{\sum_{k \le q} \binom{d}{q}}$ is indexed by a multi-index

$$\mathcal{J} = \{J = (i_1, \ldots, i_k) : i_j \in [d], k \le q\}.$$

such that:

$$[\Phi(\mathbf{x})]_J = \prod_{i \in J} \mathbf{x}_i.$$

We can think of learning a polynomial over $\mathbf{x}$ as learning a linear function over the feature vector $\Phi(\mathbf{x})$: Namely:

$$p(\mathbf{x}) = \mathbf{w} \cdot \Phi(\mathbf{x}).$$

Where $\mathbf{w}$ is a vector of the coefficients of the polynomial p.

So a first solution to this problem is to consider the labeled feature vectors $(\Phi(\mathbf{x}_i), y_i)$ and learn a linear classifier in the embedded space $\mathbb{R}^{|\mathcal{J}|}$

How would the above idea scale with degree of the polynomial. Suppose we run the perceptron in order to learn the coefficients:

- The output vector is of size $|\mathcal{J}| = O(d^{q+1})$ – so merely storing the output vector is exponential in q.
- At each iteration, we need to perform a scalar product between $\mathbf{w}_i$ and $\mathbf{x}_i$ which again requires $O(d^{q+1})$ computations (the dimension of the vectors).

Overall the computational complexity deteriorates exponentially in the degree of the polynomial. On the

other hand, the expressiveness of the class to be learnt scales, so we don't want to give up so easily. The first problem, storing the output vector, can be handled quite easily actually. Note that at each iteration of the algorithm we have that $\mathbf{w}_i \in \text{span}\{\mathbf{x}_j\}_{j \le i}$. Therefore, instead of representing $\mathbf{w}_i$ as a vector in $\mathbb{R}^{\binom{d}{q}}$ we can just store the coefficient of the m sample vectors. Namely, if

$$\mathbf{w}_{i-1} = \sum_{j \le i-1} \alpha_j \Phi(\mathbf{x}_j).$$

We store $\alpha_i = \frac{1}{2}(y_i - \hat{y_i})$ and write

$$\mathbf{w}_i \to \mathbf{w}_{i-1} - \alpha_i \Phi(\mathbf{x}_i).$$

So instead of storing a $\mathbb{R}^{\mathcal{J}}$ vector, we just store m coefficients We still need to know how to compute $\hat{y_i} = \text{sgn}(\mathbf{w}_i \cdot \Phi(\mathbf{x}_i))$ which takes $O(d^{q+1})$ time, though. Note that

$$\mathbf{w}_i \cdot \Phi(\mathbf{x}_i) = \sum \alpha_j \Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}_i).$$

To simplify notations, let $K \in M_{m \times m}$ be a matrix (dependent on the sample) such that

$$K_{j,i} = \Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}_i).$$

Then we can write

$$\mathbf{w}_i \cdot \Phi(\mathbf{x}_i) = \bar{\alpha}_i K_i.$$

where $\bar{\alpha}_i = (\alpha_1, \alpha_2, \ldots, \alpha_i)$ and K_i is the i-th column of the matrix K.

So, if we have a procedure that calculates the scalar product $\Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}_i)$, for each two data points, and produces the matrix K, then we can calculate $\hat{y}_i$ efficiently. In general, it is hard to compute $\Phi(\mathbf{x}_j) \cdot \Phi(\mathbf{x}_i)$. But suppose we have a different embedding. Instead of setting $[\Phi(\mathbf{x})]_J = \prod_{i \in J} \mathbf{x}_i$ Consider the following embedding (again in the space of polynomials:

$$[\Phi(\mathbf{x})]_J = \sqrt{\binom{d}{|J|}} \prod_{i \in J} \mathbf{x}_i.$$

Then,

$$\begin{aligned}
\phi(\mathbf{x}_1) \cdot \phi(\mathbf{x}_2) &= \sum_{J \in \mathcal{J}} \binom{q}{|J|} \prod_{i \in J} (x_1)_i \cdot (x_2)_i \\
&= \sum_{k=1}^{q} \binom{q}{k} \sum_{|J|=k} \prod_{i \in J} (x_1)_i \cdot (x_2)_i \\
&= \sum_{k=1}^{d} \binom{q}{k} \left(\sum_{i=1}^{d} (x_1)_i \cdot (x_2)_i \right)^k \\
&= \sum_{k=1}^{d} \binom{q}{k} (x_1 \cdot x_2)^k \\
&= (1 + x_1 \cdot x_2)^q.
\end{aligned}$$

Note that the last equation can be computed efficiently.

We can obtain the following variant of the Perceptron:

Algorithm 2 Kernelized Perceptron

Input: A sample $S = \{(\mathbf{x}^{(i)}, y_i)\}_{i=1}^m$.
Intialize: $\bar{\bar{\alpha}} = 0 \in \mathbb{R}^m$, $K_{i,j} = \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}_j)$.

For i=1 to m
Set $\hat{y}_i = \text{sgn}(\bar{\alpha} K_i)$ $\%K_i$ is the i-th column of K
Update

$$\bar{\alpha}_i = \frac{1}{2}(y_i - \hat{y}_i) \qquad \text{i.e.}\%\mathbf{w}_i = \begin{cases} \mathbf{w}_{i-1} & y_i = \hat{y}_i \\ \mathbf{w}_{i-1} - \hat{y}_i \Phi(\mathbf{x}_i) & y_i \neq \hat{y}_i \end{cases}$$

End For

5.2.1 The Kernel trick

The above idea can be generelized to other non-linear embeddings. Namely, suppose we have a feature space in high dimensional, and an embedding $\mathbf{x} \to \Phi(x)$ such that, the scalar product

$$k(\mathbf{x}_1, \mathbf{x}_2) = \Phi(\mathbf{x}_1) \cdot \Phi(\mathbf{x}_2)$$

can be computed efficiently. The function k is then called a *kernel function*. Given a sample, the matrix $K_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$ is called, the *kernel matrix*, and we an apply algorithm 2 readily, with any kernel function and kernel matrix. Note, though, that we make the *margin* assumption. The following are example for kernel functions:

- $k(x_i, x_j) = (1 + x_i \cdot x_j)^q$ (polynomial kernel).
- $k(x_1, x_1) = e^{-\frac{\|x_1 - x_2\|^2}{2\sigma^2}}$ (Radial basis kernel).
- $k(x_i, x_j) = \frac{1}{1 - \nu x_i \cdot x_j}$ $(0 < \nu < 1)$

Remark on the sample complexity Note that we did not discuss the sample complexity. In general, because we embed in a high-dimensional space, it may seem as if the sample complexity also scales. It turns out, though, that the margin assumption also can be used to inhibit the sample complexity. We will discuss this further in future lectures. But, surprisingly, the overall sample complexity of the problem turns out to scale with the margin, and *not* with the dimensionality of the feature space.

References