CS269: Machine Learning Theory
Lecture 6: Online Learning
October 13, 2010

Lecturer: Jennifer Wortman Vaughan
Scribe: Shilpa Vijayan, Petch Wannissorn

1 Online Learning Setting

In this class, we shift to a different learning model called the online learning setting. So far we have been considering the batch learning setting where an algorithm is first given a set of training data and it comes up with a function that performs well on new data drawn from the same distribution. The goal of Batch learning is to receive all data at once and select the best function. However it may be unrealistic to expect to get a representative data set up front and come upon a function that performs well on all data that comes thereafter. In an online learning setting, we run an algorithm to learn a function over time and update it when it gets more data. For example, consider the spam filtering problem. The spam filter updates its prediction function when the email user corrects its mistake, e.g. by explicitly marking an email that got past the spam filter as spam.

Basic Online Setting

In a basic online setting, an algorithm proceeds in a series of rounds.
At each round t,

1. The learning algorithm sees a new example $\vec{x_t}$.
2. The algorithm predicts a label $\widehat{y_t}$ for this example.
3. After the prediction, the true label y_t is revealed.
4. Algorithm makes a mistake if the predicted label is not equal to the true label i.e. $y_t \neq \widehat{y_t}$
5. Update the function used to make the prediction.

In this learning setting unlike the PAC learning model, no distributional assumptions are made about the $\vec{x_t}$. Hence the online learning setting is adversarial in the sense that no distributional assumptions are being made and there are no restrictions on where the examples come from. We can think of the examples as being generated by an adversary and, because of this, it turns out there are a lot of connections between online learning and game theory.

There are 2 goals of learning in the online setting:

1. Minimize the number of mistakes - Here we want to bound the total number of mistakes made by the algorithm such that the ratio

$$\frac{\#\,of\ mistakes}{\#\,ofrounds}$$

tends to zero as number of rounds tend to infinity.

This is analogous to the realizable setting where we assumed that there is a perfect target function you can apply to get a perfect labeling each time and the goal was to have the error measure of our hypothesis function go to zero as the number of examples grows.

2. Minimize regret - This is analogous to the unrealizable setting where we minimize the difference between the number of mistakes the algorithm makes and the number of mistakes made by the best predictor or comparator in a class of functions. In short, we want the ratio

$$\frac{(\#\,of\,mistakes - \#\,of\,mistakes\,by\,comparator)}{\#\,of\,rounds}$$

tends to zero as number of rounds tend to infinity.

The minimizing regret approach is more realistic and useful in real world applications where we cannot assume perfection and noise free data. However, we will concentrate on the first goal, mistake minimization, for today's lecture. To get intuition about the learning model and types of things that can be learned we will establish some simple upper and lower bounds and then look at some specific functions like the linear threshold functions.

2 Upper Bound

We will derive an upper bound that can be applied to any finite class of functions one is trying to learn. In this respect, consider the halving algorithm. The halving algorithm is an extremely simple algorithm although it suffers from the disadvantages of being computationally infeasible and from having to assume the existence of a perfect target, which is sometimes unrealistic. To derive the upper bound we introduce the concept of a version space. We define version space as the set of all functions in concept class C that is consistent with all examples seen so far i.e.

Version Space = $\{$all consistent functions in $C\}$.

As the algorithm learns the true labels of more examples, some functions will be dropped from the version space if they become inconsistent.

Halving Algorithm

Theorem 1. *Let C be any finite function class. Assume* $\exists c \in C$ *such that* $\forall t, y_t = c(\vec{x_t})$. *Then the number of mistakes made by the Halving Algorithm is no more than* $\log |C|$

We define the halving algorithm as the following.
Assume $\exists c \in C$ such that $\forall t, y_t = c(\vec{x}_t)$
or in words, assume there exists some perfect target function c in some finite class of functions C such that, for all t where t denotes a round, $y_t = c(\vec{x_t})$, then

At each round t,

1. Predict label $\widehat{y_t}$ to be the same as is chosen by the majority of functions in the current version space.

2. Compute the new version space.

Analysis of this algorithm is as follows:
After zero mistakes, the size of the version space is upper bounded by the size of the class $\mathcal{C}$ i.e.

$$|VS| \leq |\mathcal{C}|$$

After 1 mistake, we can say that since the majority of the functions in the current version space labeled the point wrong, we can eliminate at least half of them. Hence after eliminating half of the functions from the version space (VS) we have

$$|VS| \leq \frac{|\mathcal{C}|}{2}$$

After 2 mistakes, and eliminating another half of VS we have

$$|VS| \leq \frac{|\mathcal{C}|}{4}$$

More generally we can write,
After k mistakes

$$|VS| \leq \frac{|\mathcal{C}|}{2^k}$$

We assumed that there is at least 1 perfect function in class hence $1 \leq |VS|$. Combining the two expressions for the upper and lower bounds we have,

$$\begin{aligned} 1 \leq |VS| &\leq \frac{|\mathcal{C}|}{2^k} \\ 2^k &\leq |\mathcal{C}| && \text{combining the upper and lower bounds} \\ k &\leq log\,|\mathcal{C}| && \text{taking log on both sides} \end{aligned}$$

This result has the log$|\mathcal{C}|$ dependence of the bound and reflects the Occam's razor principle that the bigger the class we start with, the more expensive it is to learn the class. As an example consider the class of monotone disjunctions where $|\mathcal{C}| = 2^n$ for n features. The number of mistakes is bounded by n for this class. However as mentioned above, this algorithm is computationally infeasible as it requires setting up a version space of 2^n functions and computing a majority label by evaluating 2^n functions at every step of the algorithm.

3 Lower Bound

We first show the lower bound for deterministic algorithms and see that there is a dependence on VC-dimension in this model as well. Assuming that there is a perfect target $c \in \mathcal{C}$ and VC-dimension of that class is d i.e. $d = VC(\mathcal{C})$, then in this case, you can force a deterministic algorithm $\mathcal{A}$ to make at least d mistakes.

The adversary knows the prediction the algorithm will make each time and has the power to choose the sequence of $\vec{x_t}$ points(as there are no distributional assumptions) and also the label to coerce the deterministic algorithm to make a mistake. The definition of the VC-dimension says that there is some set of points $\vec{x_1}, ..., \vec{x_d}$ that is shattered by $\mathcal{C}$. So for the first d rounds, the adversary gives the algorithm the first d points shattered by C. The adversary can choose any labeling for d and there will still be some $c \in \mathcal{C}$ consistent with that labeling, so it can cause the algorithm to make a mistake on each of these rounds while ensuring that some function is still consistent.

If the algorithm is randomized, the adversary can cause at least $d/2$ mistakes *on expectation* by choosing the labels of these d shattered points randomly.

4 Properties of Linear Threshold Functions

Consider the specific case of learning linear threshold functions in an online learning setting. The class of labels is denoted as $y \in \{-1, 1\}$ .Note the change of notation from {0, 1} to {-1, 1}. It is still a binary classifier but {-1, 1} make the derivations simpler.
Consider a two dimensional plane with a linear separator through the plane separating the positive and negative regions.

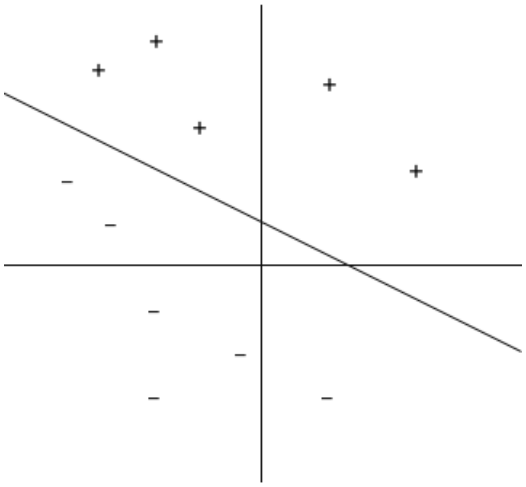

Figure 1

The linear separator is represented by the following function:

$$\vec{w} \cdot \vec{x} + w_0 \geq 0, \text{ predict 1}$$
$$\vec{w} \cdot \vec{x} + w_0 < 0, \text{ predict -1}$$

where $\vec{w}$ is the weight vector,
$\vec{x}$ is the feature vector of the example,
and w_o is a scalar quantity added to the function when the linear separator does not pass through the origin.

It is generally easier to assume that linear separators that separate the positive and the negative regions, go through the origin. We can convert from a general class of linear separators to this simple class by adding

one more dimension to the feature vector $\vec{x}$.
Let $x_o = 1$ be the dummy feature added to the set of features $\vec{x}$ for every example. Then the above function is now written as,

$$\begin{aligned}\vec{w} \cdot \vec{x} + w_0 &= \vec{w} \cdot \vec{x} + w_0 x_0 \\ &= \vec{w'} \cdot \vec{x'}\end{aligned}$$

We can see that $\vec{w'} \cdot \vec{x'}$ is just $\vec{w} \cdot \vec{x}$ with an extra component. If $\vec{w}$ was n-dimensional then $\vec{w'}$ is (n+1)-dimensional. This allows a general class of functions to map to a more simple class just by raising the dimension of the feature set by one. Now we can restrict our attention to threshold functions that passes through the origin since we know that it can be applied more generally to functions that do not pass through the origin (i.e. by a simple addition to the feature vector).

4.1 Margin

The margin of a point is defined as the distance of the point from the separating threshold function. Let γ_i be defined as the margin of point $\vec{x_i}$ then it is the distance from $\vec{x_i}$ to the separating threshold function. (We have drawn the margin in 2 dimensions in the figure below, but it is defined in the same way in higher dimensions too.) Intuitively, the margin can be viewed as an indication of the level of confidence about the labeling of the point. A large margin indicates greater confidence in labeling it. A smaller margin provides lesser confidence.

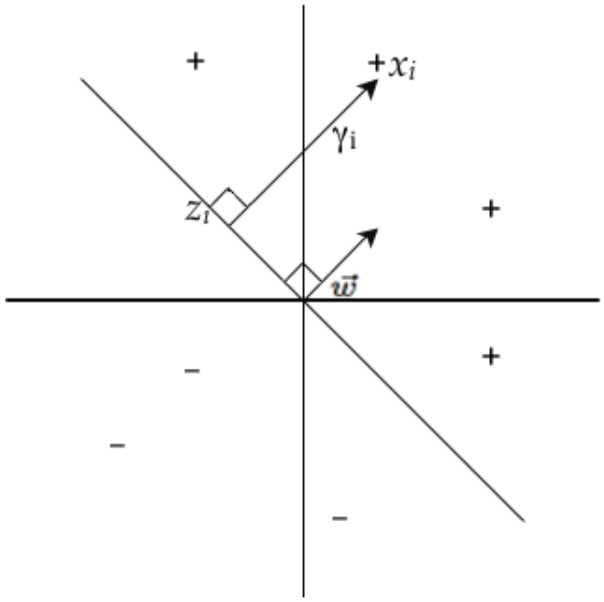

Figure 2

The linear separator is defined by the function,

$$\vec{w} \cdot \vec{x} \geq 0, \text{ predict 1}$$
$$\vec{w} \cdot \vec{x} < 0, \text{ predict -1}$$

where $\vec{w}$ is the normal vector to the separating plane.

For any point on the linear separator, $\vec{w} \cdot \vec{x} = 0$. Also assume $||\vec{w}|| = 1$. This is without loss of generality since scaling $\vec{w}$ will result in the same threshold. We define $\vec{z_i}$ as the projection of $\vec{x_i}$ on the linear separator i.e. the closest point to $\vec{x_i}$ on the linear separator. $\vec{x_i} - \vec{z_i}$ is orthogonal to the linear separator. Also note that $\vec{x_i} - \vec{z_i}$ is parallel to $\vec{w}$.
We now write the equation for the margin of the points that are labeled positive as,

$$\vec{x}_i - \vec{z}_i = \gamma_i \vec{w} \text{ (from the assumption } ||\vec{w}|| = 1) \tag{1}$$

$$\vec{z}_i = \vec{x}_i - \gamma_i \vec{w} \tag{2}$$

$$\vec{z}_i \cdot \vec{w} = 0 \text{ } (\vec{z}_i \text{ is on the decision boundary}) \tag{3}$$

$$\vec{x}_i \cdot \vec{w} - \gamma_i \underbrace{(\vec{w} \cdot \vec{w})}_{=1} = 0 \text{ (combining (2) and (3))} \tag{4}$$

For points that are labeled negative, the derivation is very similar. Instead of (1), we have

$$\vec{x}_i - \vec{z}_i = -\gamma_i \vec{w} \tag{5}$$

We then get

$$\gamma_i = \begin{cases} \vec{x}_i \cdot \vec{w}_i & \text{for points that are labeled positive} \\ -\vec{x}_i \cdot \vec{w}_i & \text{for points that are labeled negative} \end{cases}$$

or

$$\text{Margin } \gamma_i = y_i \vec{x}_i \vec{w}$$

5 The Perceptron Algorithm

The Perceptron algorithm is an extremely simple algorithm for learning a linear threshold function and works fairly well in a number of cases. The algorithm is given as follows:

1. Initialize the weight vector $\vec{w_1}$ to be all zeroes.

2. In each round, for each example represented by $\vec{x_t}$,
   Predict 1 if $\vec{w_t} \cdot \vec{x_t} \geq 0$,
   Predict 0 otherwise.

3. Update the weight vector $\vec{w}$ in case we make a mistake i.e.
   If $sign(\vec{w_t} \cdot \vec{x_t}) \neq \vec{y_t}$ then,
   update $\vec{w}_{t+1} = \vec{w}_t + y_t \cdot (\vec{x}_t / ||\vec{x}_t||)$
   otherwise $\vec{w}_{t+1} = \vec{w}_t$

The general intuition behind the algorithm is that every time it makes a mistake on a positive example, it shifts the weight vector toward the input point, whereas every time it makes a mistake on a negative point, it shifts the weight vector away from that point.

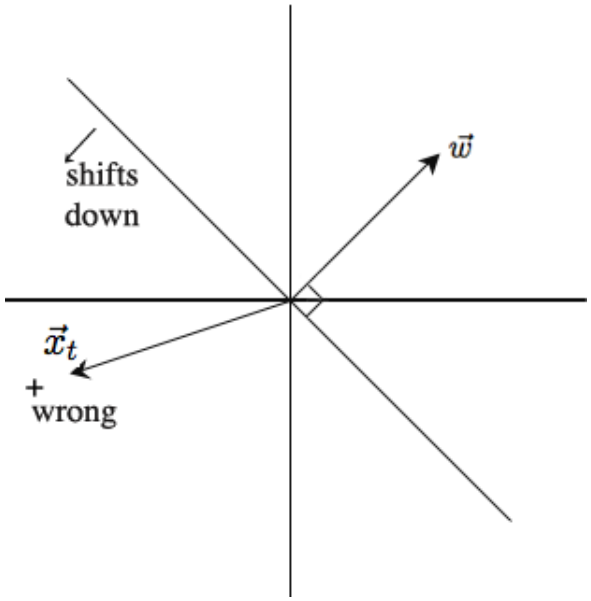

Figure 3

Theorem 2. *Suppose there exists a perfect weight vector $\vec{u}$, where $||\vec{u}||$ = 1 and suppose there's a value $\gamma > 0$, such that $\forall t, y_t \cdot (\frac{\vec{x_t}}{||\vec{x_t}||}) \cdot \vec{u} \geq \gamma$, then the number of mistakes made by the Perceptron algorithm is at most $(1/\gamma)^2$.*

This is effectively stating that if you can assume that your (normalized) data has a margin of at least γ then the number of mistakes made is at most $(1/\gamma)^2$. Note that the algorithm does not depend on γ, so you do not necessarily know what γ is in advance for this to hold. We will go over the full proof in the next class.