CS269: Machine Learning Theory
Lecture 8: Multiplicative Weights
October 20, 2010

Lecturer: Jennifer Wortman Vaughan
Scribe: Chien-Ju Ho, Liang-Chieh Chen

In today's lecture, we are going to talk about one more example of online learning: Winnow algorithm, which has the same assumption as Perceptron algorithm that there exists a perfect target function. In the second part of the class, we will talk about regret minimization with respect to some comparators.

1 Winnow Algorithm

In the previous lecture, we introduced Perceptron algorithm, which deals with online learning problems by updating the weight function according to the history of mistakes. In particular, the weight function is updated additively: $\vec{w_{t+1}} = \vec{w_t} + y_t\vec{x_t}$. We now describe Winnow algorithm, which is similar to Perceptron algorithm. However, Winnow algorithm updates the weight function multiplicatively.

Now, we formally describe the algorithm. Winnow algorithm maintains a weight vector $\vec{w_t}$. Let $w_{i,t}$ = the weight on feature i at round t, and $x_{i,t}$ = the ith component of $\vec{x_t}$. Suppose $y_t \in \{0, 1\}$, and $\vec{x_t} \in \{0, 1\}^n$ is a binary string.

The Winnow Algorithm

1. Initialize all the weights to one: $w_{1,1} = w_{2,1} = \ldots = w_{n,1} = 1$
2. For each example $\vec{x_t}$,
   - output 1, if $\vec{w_t} \cdot \vec{x_t} \geq n$.
   - output 0, otherwise.
   - if the algorithm makes a mistake,
     - If $y_t = 1$, then $\forall i$ such that $x_{i,t} = 1,\ w_{i,t+1} \leftarrow w_{i,t}(1 + \epsilon)$
     - If $y_t = 0$, then $\forall i$ such that $x_{i,t} = 1,\ w_{i,t+1} \leftarrow \frac{w_{i,t}}{1+\epsilon}$
     - In both cases ($y_t = 1$ or 0), $\forall\ x_{i,t} = 0,\ w_{i,t+1} \leftarrow w_{i,t}$
   - if there is no mistake, then $\vec{w}_{t+1} \leftarrow \vec{w_t}$

Note that in the Winnow algorithm, ϵ is a parameter. Intuitively, if we make a mistake on a positive example, we increase the weights of the features for $x_{i,t} = 1$. Similarly, if we make a mistake on a negative example, we will decrease the weights of all the $x_{i,t}$ that contribute to $\vec{w}_{i,t} \cdot \vec{x}_{i,t}$. We show that the Winnow algorithm grows (updates) faster than Perceptron algorithm, and Winnow algorithm will make fewer mistakes than the Perceptron algorithm, when the number of relevant feature is much less than the total number of features.

1.1 Learning (Monotone) Disjunctions

We consider an example of learning monotone disjunctions using Winnow algorithm. We show the algorithm will make at most $\mathbb{O}(r \log n)$ mistakes, where n is the number of variables. Note: monotone disjunction means there is no negative literals in the disjunction. This constraint can be easily relaxed using the following trick: for each variable i, add $\neg i$ into the variables.

Theorem 1. *Suppose that the data sequence can be perfectly labeled by some monotone disjunction of r variables. Then the Winnow algorithm with $\epsilon = 1$ makes at most $2 + 3r(1 + \log n)$ mistakes.*

Proof: In this proof, we show the mistake bound on positive examples and the mistake bound on negative examples separately. The results are then combined to yield the overall bound.

- Bound the number of mistakes on positive examples ($y_t = 1$)
  Observations:

  1. When we make a mistake and $y_t = 1$, at least one *relevant weight*, the weight of relevant feature, is doubled.
  2. The weights of relevant features never decrease.

  The above observations are straightforward. In observation 1, if the weights of all relevant features are not doubled[1], y_t should be 0 since y_t is the disjunction of all relevant features. In observation 2, if the weights of relevant features decrease, it means some relevant feature $x_{i,t} = 1$ when $y_t = 0$, which is impossible by definition of a disjunction.

  Next, we are going to derive the mistake bound on positive examples. We first consider about single relevant feature. Let l be the number of times the weight of feature i has been doubled, then the weight of i will be 2^l. In addition, the weight can only be doubled when they are smaller than n. If $w_{i,t} \geq n$ and $x_{i,t} = 1$, there will be no mistake. If $w_{i,t} \geq n$ and $x_{i,t} = 0$, the weight will not be updated according to the algorithm. Therefore we can conclude $2^{l-1} < n$, i.e. $l < \log n + 1$. This means the weight of each relevant feature can not be doubled for more than $\log n + 1$ times.

  Since at least one relevant weight will be doubled for each mistake on positive examples, and each wight cannot be doubled for more than $\log n + 1$ times, the number of mistakes we make on positive examples in Winnow algorithm will be less than $r(\log n + 1)$, where r is the number of relevant features.

- Bound the number of mistakes on negative examples ($y_t = 0$)
  In this proof, we use the sum of the weights, $\sum_{i=1}^{n} w_{i,t}$, as a potential function and observe how it changes over time. Before we state the proof, let's see how this sum changes when we make mistake at time t. If we make mistakes on positive examples, we will double the weight of $w_{i,t}$ if $x_{i,t} = 1$. Therefore, the sum of weight is increased by $\sum_{i:x_{i,t}=1} w_{i,t} = \vec{w_t} \cdot \vec{x_t}$. Similarly, if we make mistakes on negative examples, we will decrease the weight by half on the weight $w_{i,t}$ if $x_{i,t} = 1$. Therefore the sum of weight is decreased by $\frac{1}{2} \sum_{i:x_{i,t}=1} w_{i,t} = \frac{1}{2} \vec{w_t} \cdot \vec{x_t}$.

  Let's state some observations:

  1. Total weight of all features are initially n.
  2. Every time we make a mistake and $y_t = 1$, the sum of weights is increased by at most n.

[1] Since $\epsilon = 1$, every time we make a mistake, the weights of some features will be either doubled or be cut by half.

3. Every time we make a mistake and $y_t = 0$, the sum of weights is decreased by at least $\frac{n}{2}$.
4. The sum of weights is always bigger than 0.

Observation 1 and 4 are straightforward by definition. In observation 2, each time we make a mistake on positive example, we will increase the sum of the weight by $\vec{w_t} \cdot \vec{x_t}$. Since we make a mistake, which means we have $\vec{w_t} \cdot \vec{x_t} < n$, the sum of weights would be increased at most n. Similarly, in observation 3, we will decrease the sum of weights by $\frac{1}{2}\vec{w_t} \cdot \vec{x_t}$, Since we make a mistake, we have $\vec{w_t} \cdot \vec{x_t} \geq n$. Therefore, the sum of weights is decreased by at least $\frac{n}{2}$.

Given the observations, we are going to derive the mistake bound on negative examples. Let us first define three variables:

- P is the number of mistakes on positive examples at time t.
- N is the number of mistakes on negative examples at time t.
- $W = \sum_{i=1}^{n} w_{i,t}$ is the total weights at time t.

According to observation 4, we know that $W > 0$. In addition, the total increase on the sum of weight at time t will be less than $Pn - N\frac{n}{2}$ by observation 2 and 3. Therefore, we can conclude that $0 < W < n + Pn - N\frac{n}{2}$, which implies that $N < 2P + 2$. Therefore, the number of mistakes we make on negative examples will be less than $2r(\log n + 1) + 2$.

Combining the mistake bounds on positive examples and negative examples, we can show the mistake bound for learning monotone disjunction using Winnow algorithm will be $2 + 3r(\log n + 1)$.

□

Comparison with a naive algorithm

To provide further insight for Winnow algorithm, we compare the performance of Winnow with the performance of another simple algorithm for learning disjunctions. We then compare the mistake bounds between this algorithm and Winnow algorithm.

The Naive Algorithm

1. Start by assuming disjunction over all variables.
2. For every input $\overrightarrow{x_t}$:
   - Predict with current disjunction
   - If we make a mistake on $y_t = 0$:
     - Drop all variables i such that $x_{i,t} = 1$
   - Note: we never make a mistake on $y_t = 1$.

Each time we make a mistake, at least one variable will be dropped. In addition, an adversary could force us to drop only one variable at a time by giving us input points in which $x_{i,t} = 1$ for only one i. Therefore the mistake bound for this algorithm is $n - r$, and this bound is tight. Let's compare this bound with the bound of Winnow algorithm $\mathbb{O}(r \log n)$. When the number of relevant features r is close to the number of total features n, e.g. $r = \frac{n}{2}$, than the mistake bound of Winnow algorithm ($\mathbb{O}(n \log n)$) will be worse than the bound of the naive algorithm ($\mathbb{O}(n)$). However, if r is much smaller than n Winnow algorithm ($\mathbb{O}(r \log n)$) outperforms the naive algorithm ($\mathbb{O}(n)$).

1.2 Learning Majority Functions

Recall the class of majority functions that we discussed in the previous lecture. These functions take on a value 1 if the majority of a specific set of r features are 1, and 0 otherwise. In the previous lecture, we showed a mistake bound of rn for learning majority functions using the Perceptron algorithm. While we will not prove it here, it can be shown that Winnow can learn majority functions with a mistake bound of $2r^2 \log n$. Clearly, we can see that the Winnow algorithm again gives a better mistake bound when the number of relevant features is much smaller than the number of total features.

2 Learning from Expert Advice

So far we have been assuming that there exists a perfect target function. In the next few lectures, we will be discussing online learning algorithms in *adversarial* settings in which we make no statistical assumptions at all about the data.

We begin by introducing the problem of learning from expert advice. Suppose that you are interested in making a sequence of predictions based on advice from several "experts." In this context, an expert could be a weather forecaster, if you are interested in tomorrow's weather, or a financial analyst, if you are concerned with the stocks you own. More generally, experts could be individual features, or individual learning algorithms, or any other predictors you have. At each time step, you must choose an expert to follow. There is loss associated with each expert, and you receive the loss of the expert you follow.

Formally, the setting can be described as follows.

Learning from Expert Advice

1. There are n experts.
2. At each round t,
   - The algorithm chooses an expert.
   - Each expert i suffers loss $l_{i,t} \in [0, 1]$
   - The algorithm suffers loss of the expert it chose.

Note that we do not make any assumption about the quality of the experts. That is, the loss ranges from 0 (perfect) to 1 (lousy). If we are doing binary classification, the loss could be 1 if the expert makes an error or 0 otherwise. But we define it much more generally here. Also note that we cannot make absolute guarantees about the quality of our predictions, since all experts might have high loss. Instead, we hope to perform as well as the best expert so far. To be concrete, we define regret as follows.

$$Regret = total\ loss\ of\ algorithm - \min_{i \in 1,\ldots,n} \sum_{t=1}^{T} l_{i,t}$$

Our goal is to minimize the regret. In particular, we would like it to be sublinear in the number of rounds T so that our *average* regret goes to 0 over time.

It is easy to see that this goal is impossible to achieve if we restrict our attention to deterministic algorithms. An adversary, trying to make the regret large, could set $l_{i,t} = 1$ for whichever expert i we choose, and $l_{j,t} = 0$ for all $j \neq i$. In this case, at round T, the total loss of the algorithm would be T, and since there

would be at least one expert having loss $\leq \frac{T}{n}$, we have $regret > T - \frac{T}{n}$, which is linear in T, not what we want.

Therefore, instead of choosing a single expert, we allow the algorithm to choose a probability distribution over experts (the weights), and define the loss that the algorithm receives to be the expected loss according to this distribution.

Modified Learning from Expert Advice

1. There are n experts.
2. At each round t,
   - The algorithm chooses weights $\vec{w_t}$.
   - The adversary chooses losses $\vec{l_t}$.
   - The algorithm suffers loss $\vec{w_t} \cdot \vec{l_t}$.

   Then the Regret is

$$\sum_{t=1}^{T} \vec{w_t} \cdot \vec{l_t} - \min_{i \in 1,\ldots,n} \sum_{t=1}^{T} l_{i,t}.$$

In the next lecture, we will show that there is an algorithm that achieves a regret of O($\sqrt{T \log n}$).