CME 323: Distributed Algorithms and Optimization, Spring 2015
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Databricks and Stanford.

Lecture 13, 5/11/2015. Scribed by Zhiang Hu (Harvy), Benoît Dancoisne, Tarek Berro.

13.1 The perceptron algorithm

In this section, we will consider two linearly separable classes with margin γ, i.e. $\gamma = \min_{x \in S} |w^\star x|$ with $||w^\star||_2 = 1$ (that is γ is the minimum distance between a point and the hyperplane defined by $w^\star$). We also assume that each data point x has been normalized such that $||x||_2 = 1$.

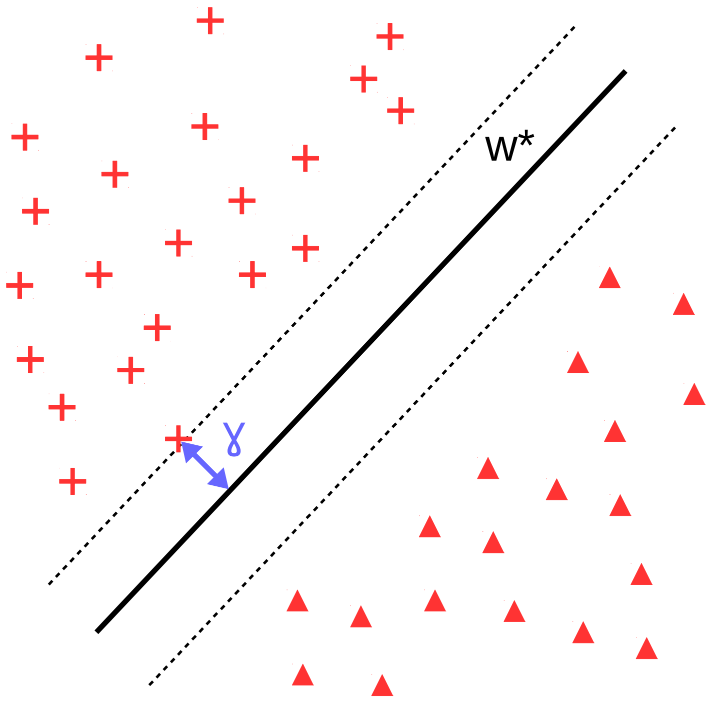

Figure 1: Linearly separable sets with margin γ

Recall the algorithm of the perceptron:

Algorithm 1 Perceptron algorithm

```
procedure PERCEPTRON
    for each node $x_i \in Data$ do
        if $w_t^T x_i > 0$ then
            Predict positive label
        else
            Predict negative label
        end if
        if wrong label then
            if true label is positive then
                $w_{t+1} = w_t + x_i$
            else
                $w_{t+1} = w_t - x_i$
            end if
        end if
    end for
end procedure
```

The number of mistakes made by this algorithm in a streaming context can be bounded as follows:

Theorem 13.1 *Under the assumption of linear separability with margin γ, the number of mistakes made by the perceptron algorithm is at most $1/\gamma^2$*

To prove the above statement, we first need to prove the two following claims:

Claim 13.2 *After every mistake, we have $w_{t+1}^T w^\star \geq w_t^T w^\star + \gamma$.*

Proof: On a positive mistake, $w_{t+1}^T w^\star = w_t^T w^\star + x^T w^\star \geq w_t^T w^\star + \gamma$ as $x^T w^\star > \gamma$
On a negative mistake, $w_{t+1}^T w^\star = w_t^T w^\star - x^T w^\star \geq w_t^T w^\star + \gamma$ as $x^T w^\star \leq \gamma$
Therefore $w_{t+1}^T w^\star \geq w_t^T w^\star + \gamma$ after any mistake. ■

Claim 13.3 $||w_{t+1}||_2^2 \leq ||w_t||_2^2 + 1$

Proof: On a positive mistake, $||w_{t+1}||_2^2 = ||w_t||_2^2 + 2w_t^T x + ||x||_2^2 \leq ||w_t||_2^2 + 1$ since $w_t^T x \leq 0$ and $||x||_2^2 = 1$.
On a negative mistake, $||w_{t+1}||_2^2 = ||w_t||_2^2 - 2w_t^T x + ||x||_2^2 \leq ||w_t||_2^2 + 1$ since $w_t^T x > 0$ and $||x||_2^2 = 1$.
Therefore $||w_{t+1}||_2^2 \leq ||w_t||_2^2 + 1$ after any mistake. ■

We can now prove theorem 13.1

Proof of Theorem 13.1: Suppose that M mistakes have been made. Then as we start with $w_0 = 0$ Claim 13.2 yields

$$w_M^T w^\star \geq M\gamma$$

and Claim 13.3 yields

$$||w_M||_2^2 \leq M$$

Putting those together we get

$$M\gamma \leq w_M^T w^\star \underset{C.S.}{\leq} ||w_M|| \leq \sqrt{M}$$

and thus

$$M \leq \frac{1}{\gamma^2}$$

■

13.2 Interface between theory and applications

Parallel computing uses different primitive from those usually encountered in sequential programming such as for and $while$ loops, if and $switch$ statements. Those are used in different contexts:

Embarrassingly parallel problems For this kind of problem we only need the Map primitive (in any distributed programming framework, not just Spark)

All to One and One to All communications The primitives used here are *Reduce* (the result ends up on one machine, with an almost optimal cost equal to the cost on one machine divided by the number of machines), *Broadcast* (whose communication cost is $\log_b(k)$ for a bandwidth b on k machines using a tree-like propagation) and *AllReduce* (the result ends up on all machines, essentially a *Reduce* followed by a *Broadcast*)

All to All communication The primitive used here are *Join*, *ReduceByKey*, *GroupByKey* (used when the reducing function is not associative and commutative), and *Sort*. (Again, these primitives are found in any distributed programming framework, not just Spark)

13.3 The ADMM algorithm

Consider the following optimization problem:

$$min_x F(x) = \sum_i f_i(x)$$

- If all the f_i are continuous and convex, we have access to the subgradients and can use gradient descent algorithms
- If the f_i are strongly convex, we can use parallel stochastic gradient descent. In order to do this we need to shuffle the data once, but as this implies sorting the data the shuffle size is of order $\mathcal{O}(n)$ (which we would like to avoid)

- If the only assumption we make on the f_i is convexity we can still use ADMM (Alternating Direction Method of Multipliers). This algorithm is particularly useful in cases where

    1. we have no access to subgradients
    2. each function is convex
    3. we want to avoid communication dependence on n

Note: ADMM is actually rarely used because of its poor convergence rate. It brings down the error in objective by only $1/\sqrt{k}$ with each iteration k. Its advantage, however, is that it's "apologies free".

In ADMM, each machine j deals with a subset F_j of functions f_i in the original sum, and for each iteration k, the local parameter x_j is updated as follows:

$$x_j^{k+1} = \arg\min_{x_j}(f_j(x_j) + y_j^{kT} \cdot (x_j - \bar{x}^k) + \frac{p}{2}||x_j - \bar{x}^k||_2^2)$$

$$y_j^{k+1} = y_j^k + p(x_j^{k+1} - \bar{x}^{k+1})$$

Where p is the penalty parameter, $\bar{x}^k = \frac{1}{m}\sum_j x_j^k$, m is the number of machines. y_j^k is an auxiliary variable that indicates how close we are from the consensus.

Note that at each iteration, the only thing communicated through the network is $\bar{x}^{k+1}$ so that x_i^{k+1} can be computed locally. This is a one to all communication.

13.4 Note on how AllReduce works

We should also note that, to broadcast a vector we do not simply send a message from one machine to all the $m-1$ other machines directly, since the network bandwidth b of one machine could be the bottleneck. Instead, we should first send the message to as many machines as possible (that is b machines) and then send the message to $\mathcal{O}(b^2)$ machines from the $b+1$ machines and so on. This takes $\mathcal{O}(log_b(m))$ steps and the total communication time is $\mathcal{O}(d \cdot log_b(m))$ if $x \in \mathbb{R}^d$, and d is the dimension of x.