Lecture 16: Solving LPs with multiplicative weights

Jerry Li

February 28, 2025

1 Definitions

Multiplicative weights. Recall the (general) learning from experts problem from last lecture. Here, the setup is the following online learning problem. There are n experts, labeled $\{1, \dots, n\}$, and for $t = 1, \dots, T$ rounds, we repeat the following procedure.

1. The player specifies a distribution over experts $p_t : [n] \to \mathbb{R}_{\geq 0}$.
2. The adversary reveals a loss $\ell_t : [n] \to [-1, 1]$. Crucially for later, this loss is allowed to depend on p_t.
3. The player pays loss

$$\mathbb{E}_{i \sim p_t}[\ell_t(i)] = \sum_{i=1}^{m} p_t(i)\ell_t(i) = \langle p_t, \ell_t \rangle \ ,$$

which we can think of as the expected loss of the player, if the player selects a random expert i from distribution p_t, and paying the corresponding loss.

The goal of the player is to minimize regret, defined as

$$\mathrm{Reg}(T) = \sum_{t=1}^{T} \mathbb{E}_{i \sim p_t}[\ell_t(i)] - \min_{i \in [n]} \sum_{t=1}^{T} \ell_t(i) \ ,$$

that is, regret measures how much worse the player performs relative to the best performing expert in retrospect. The main result that we sketched out last lecture is the following:

Theorem 1.1. *The multiplicative weights update achieves regret* $\mathrm{Reg}(T) \leq O(\sqrt{T \log n})$.

Linear programming. On the other hand, we had previously also considered the problem of solving linear programs. Recall that a linear program is specified by:

- n free, real-valued variables $x_1, \dots, x_n$.
- A linear objective $c \in \mathbb{R}^n$.
- m linear constraints, specified by $a_1, \dots, a_m \in \mathbb{R}^n$ and values $b_1, \dots, b_m \in \mathbb{R}$.

Then, the objective is to solve the following:

$$\max \langle c, x \rangle \text{ s.t. } \langle a_i, x \rangle \geq b_i \text{ for all } i = 1, \dots, m \ .$$

Recall that in general, we might also want to have other types of linear constraints, including $\langle a_i, x \rangle \leq b_i$ constraints and $\langle a_i, x \rangle = b_i$ constraints, however, we can encode these additional types of constraints using the type of constraint given here. Similarly, we can also do minimization rather than maximization by negating the objective function. Recall also that we often write the constraints as $Ax \geq b$, where A is the $m \times n$ matrix where the i-th row of the matrix is $a_i^\top$, and the inequality is to be interpreted coordinate-wise.

2 Preliminaries

The main result of this lecture is a method that allows us to approximately solve linear programs using multiplicative weights. Initially, this might look quite surprising: after all, MW is a method for online learning, but linear programming is just an offline optimization problem. But it turns out that online learning is a surprisingly powerful and general primitive, and can be used to encode very general optimization problems, including linear programming. However, before we get into the meat of the algorithm, we first need a couple of preprocessing steps and preliminaries.

From optimization to feasibility Rather than trying to solve for the x that minimizes some objective function, it will be easier for us to think about feasibility question: namely, is it even possible to find x satisfying $Ax \geq b$, or is this set of constraints infeasible?

It turns out that solving this feasibility problem already suffices to get a good algorithm for optimization. The trick is to, for some parameter λ, consider the following feasibility question: does there exist x so that $\langle c, x\rangle \geq \lambda$ and $\langle a_i, x\rangle \geq b_i$ for all $i = 1, \ldots, m$? This is now a feasibility problem over $(m+1)$ constraints. Moreover, if we can solve this feasibility problem for any choice of λ, then to find the value of the optimization problem, we can simply binary search over λ, performing this feasibility check at every point, to find the largest λ for which this feasibility problem is satisfiable. The resulting λ will then clearly be the solution to the original optimization problem.[1]

For the rest of this lecture, in a slight abuse of notation, let's only consider how to solve the original feasibility problem on m constraints; this is just for conciseness and to avoid carrying around additional parameters. So now, we have m constraints $a_1, \ldots, a_m \in \mathbb{R}^n$ and values $b_1, \ldots, b_m$, and our goal is to either find $x \in \mathbb{R}^n$ so that $Ax \geq b$, or determine that the problem is infeasible.

Obtaining a width bound One additional parameter our method will require is some notion of the *width* of the problem. We will assume that there is some convex set K for which we know *a priori* that if there is a feasible solution to the LP, then that solution must necessarily lie in K. For instance, a natural choice (albeit suboptimal in many settings, it turns out), is to choose K to be a large ℓ_2 ball around the origin: i.e. for some parameter R, we take

$$K = \{x \in \mathbb{R}^n : \|x\|_2 \leq R\} .$$

We can think of this as a large circumscribing ball for all possible solutions to our linear program, which allows us to restrict our search to only points in K. Given this set K, we can then define the *width* of our linear program with respect to K, denoted ρ, as

$$\rho = \max_{x \in K}\{1, \max_{i \in [m]}(|\langle a_i, x\rangle - b_i|)\} .$$

Let's unpack this definition for a second. The width is really just the maximum that any constraint can be violated by any point in our set K. We then also take a second maximum with 1 to ensure that ρ is never less than 1, which is there for technical reasons that you don't really need to worry about.

Solving a 1-constraint LP Finally, we will also need the ability to efficiently solve a linear program with one constraint subject to the constraint that $x \in K$. That is, we need to solve the following problem: given K, and given $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$, find $x \in K$ so that $\langle a, x\rangle \geq b$ if such an x exists, and output NO if no such x exists. When K is an ℓ_2-ball, this is easy: this essentially corresponds to checking the norm of the projection of 0 onto the line (technically, affine subspace) $L = \{x : \langle a, x\rangle = b\}$, which we can do easily. We leave the full details of this subroutine as an exercise to the reader. For other choices of K it's not as immediate, but it still turns out that we can do this very efficiently.

[1] Here, we are purposefully ignoring the possibility that the original value of the linear program could be infinite. It's a nice exercise to think how to deal with this possibility!

3 Solving LPs with MW

With these, we can now state our guarantee:

Theorem 3.1. *Let $\varepsilon > 0$. There is an algorithm, which, given m constraints $a_1, \dots, a_m \in \mathbb{R}^n$ and $b_1, \dots, b_m \in \mathbb{R}$, and a convex set K satisfying the conditions from the previous section, either certifies that there is no point $x \in K$ so that $\langle a_i, x \rangle \geq b_i$ for all $i = 1, \dots, m$, or finds a point $x \in K$ so that $\langle a_i, x \rangle \geq b_i - \varepsilon$ for all $i = 1, \dots, m$. The algorithm runs in time polynomial in n, m, ρ, and $1/\varepsilon$.*

That is, this algorithm efficiently finds a point which is approximately feasible for the linear program, or verifies that there is no point in K which is a feasible solution. Before we prove this theorem, we make a couple of remarks:

- As one can back out from the final analysis, assuming that solving 1-constraint LPs over K is not the bottleneck, it turns out that the overall runtime of this algorithm is actually just

$$O\left(\frac{nm\rho \log m}{\varepsilon^2}\right).$$

  So when ρ and ε are both constant, our runtime is actually linear (up to logarithmic factors) in the size of the input, since the input has size $O(nm)$.

- There are two suboptimal dependencies in this runtime: namely, the dependence on ρ and the dependence on $1/\varepsilon$. In some sense, an ideal algorithm should have no dependence on the width whatsoever! However, as we'll see, this is unavoidable for us.

- Similarly, ideally an algorithm should run in time which is polynomial in $\log 1/\varepsilon$: this would allow us to solve any LP specified with B bits of accuracy exactly in time which is polynomial in the overall size of the problem. However, it turns out for methods based on multiplicative weights, and a more general class of methods loosely termed "first order" methods, of which multiplicative weights belongs, a polynomial dependence on $1/\varepsilon$ is somewhat unavoidable.

We will specify an algorithm using the multiplicative weights framework. Note that when specifying such an algorithm, the only flexibility we have is how the adversary chooses their losses: the player's responses are fixed ahead of time by the multiplicative weights strategy. So our job is to be a productive "bad guy" for the online learning framework.

The high-level idea is that we will treat each of the m constraints as an expert. We will secretly maintain a sequence of iterates x_t (which the player doesn't actually know about), but the goal is to use the feedback from multiplicative weights to make it so that all the constraints are satisfied. We will do so by making the player's feedback tell us how to reweight the relative importance of every constraint. Ideally, we want the player to force us to put more emphasis on constraints that are violated, and relatively less emphasis on constraints which are satisfied. In the end, the hope that by doing so, all the constraints will be equally satisfied (at least, approximately).

We will do so in a slightly counterintuitive way: we will assign large loss to constraints which are *satisfied* by our current iterate, and we will assign small loss to constraints which are *violated*. The reason for this is because we want the player to put large weight on constraints which are violated: as we'll see, this will force us to pay more attention to them in the next iteration, and hopefully get closer to satisfying them. On the other hand, we can afford to put less weight on constraints which are satisfied, since they're okay for now. Recall that the multiplicative weights update works as follows: given a current set of weights w_t, our update is

$$w_{t+1}(i) = w_t \cdot e^{-\varepsilon \ell_t(i)}.$$

So this update down-weights experts with large loss, and up-weights experts with small loss. Therefore, to get the desired behavior, we need to put large loss of constraints which are satisfied.

With all of this discussion in mind, we now come to the adversary's strategy. Let T be some parameter we will set later. Then, for $t = 1, \dots, T$, we feed in the following into the multiplicative weights framework:

1. We receive a distribution from the player p_t.
2. We form the "average" constraint $a^{(t)} = \sum_{i=1}^m p_t(i)a_i$ and $b^{(t)} = \sum_{i=1}^m p_t(i)b_i$ using the weights from the player.
3. Solve the 1-constraint LP: find $x \in K$ so that $\langle a^{(t)}, x\rangle \geq b^{(t)}$ if such a point exists. If the 1-constraint solver says that this is infeasible, terminate and output that the overall LP is infeasible.
4. Otherwise, let x_t be the solution to this 1-constraint LP.
5. Form the loss vector

$$\ell_t(i) = \frac{\langle a_i, x_t\rangle - b_i}{\rho} ,$$

and give this as feedback to the multiplicative weights player.

At the end of T iterations, if the algorithm has not terminated early, output as our final solution

$$\tilde{x} = \frac{1}{T}\sum_{t=1}^{T} x_t .$$

The rest of this section is dedicated to the proof of correctness of this algorithm. First, we show that if the algorithm terminates early, then indeed the LP must have been infeasible:

Lemma 3.2. *Suppose that our algorithm ever terminates early. Then there is no point $x \in K$ satisfying $Ax \geq b$.*

Proof. We will prove the contrapositive: if there was a feasible point for the original LP, then the algorithm will never terminate early. Indeed, suppose there is $x \in K$ so that $\langle a_i, x\rangle \geq b_i$ for all $i = 1, \ldots, m$. But note that then, we have that

$$\langle a^{(t)}, x\rangle = \left\langle \sum_{i=1}^m p_t(i)a_i, x \right\rangle = \sum_{i=1}^m p_t(i)\langle a_i, x\rangle \geq \sum_{i=1}^m p_t(i)b_i = b^{(t)} ,$$

where the inequality follows by our assumption that $\langle a_i, x\rangle \geq b_i$ for all $i = 1, \ldots, m$, and since the $p_t(i)$ are all non-negative. Thus the averaged constraint will be satisfiable, and our algorithm will never terminate early. □

We can now finish the proof of the theorem:

Proof of Theorem. It now remains to prove that if the algorithm does not terminate prematurely, then the resulting $\tilde{x}$ will satisfy the properties of the theorem, i.e. that it will be an approximately feasible point. In particular, we will show that it suffices to take

$$T = O\left(\frac{\rho^2 \log m}{\varepsilon^2} .\right) \tag{1}$$

To do so, let's instantiate the regret minimization bound; this is, after all, the only tool we really have! Before this, we need to briefly verify that we can apply the regret minimization bound: the key thing that we need to check is that the loss $\ell_t(i)$ is always bounded between -1 and $+1$. Recall that $\ell_t(i) = (\langle a_i, x_t\rangle - b_i)/\rho$. But this is exactly why we divide the loss by ρ: in particular, since $x_t \in K$, we know that

$$|\langle a_i, x_t\rangle - b_i| \leq \max_{x\in K} |\langle a_i, x\rangle - b_i| \leq \rho ,$$

so indeed, the loss vector is entrywise bounded by 1 in magnitude. So we can apply the regret guarantee.

Rearranging slightly, the regret bound is equivalent to:

$$\min_{i\in[m]} \sum_{t=1}^{T} \ell_t(i) \geq \sum_{t=1}^{T}\sum_{i=1}^{m} p_t(i)\ell_t(i) - O(\sqrt{T\log m}) \ ,$$

Now, let's just plug in the definition of ℓ_t, and there is where something kind of magical happens. Multiply both sides by ρ and divide by T, and we obtain that

$$\min_{i\in[m]} \frac{1}{T}\sum_{t=1}^{T}(\langle a_i, x_t\rangle - b_i) \geq \frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{m} p_t(i)(\langle a_i, x_t\rangle - b_i) - O\left(\rho\sqrt{\frac{\log m}{T}}\right) \ ,$$

Now look at the left-hand side: this is exactly equal to $\min_{i\in[m]}\langle a_i, \tilde{x}\rangle - b_i$, by the way we defined $\tilde{x}$. On the other hand, if we consider the first term on the right-hand side, we observe that for any t, we get that

$$\sum_{i=1}^{m}(\langle a_i, x_t\rangle - b_i) = \langle a^{(t)}, x_t\rangle - b^{(t)} \ ,$$

but by the way we chose x_t, this is always non-negative. Hence the regret bound implies that

$$\min_{i\in[m]} \langle a_i, \tilde{x}\rangle - b_i \geq -O\left(\rho\sqrt{\frac{\log m}{T}}\right) \ .$$

By our choice of T in (1) and setting constants appropriately, we obtain that $O(\rho\sqrt{\frac{\log m}{T}}) \leq \varepsilon$. So we have shown that for all $i \in [m]$, we must have

$$\langle a_i, \tilde{x}\rangle \geq b_i - \varepsilon \ ,$$

which is what we wanted to show. □