CIS 320: Algorithms October 27 and November 1, 2021

Lecture 15-16

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Polynomial Weights I and II

In the next two lectures, we will develop algorithms for *prediction* problems that must be solved in an adversarial, sequential setting. These algorithms operate in an environment that we haven't studied before in this class: rather than having a problem instance fully described up front, that we must solve, we will have an environment in which our algorithm must interact dynamically. The algorithms we derive will be interesting in their own right, and are fundamental building blocks in machine learning, but as we will see, they are also useful and powerful tools for solving other algorithmic problems in a more standard setting.

As a simple example to keep in mind, consider the following toy model of predicting the stock market: every day the market goes *up* or *down*, and you must predict what it will do before it happens (so that you can either buy or short shares). You don't have any information about what the market will do, and it may behave arbitrarily, so you can't hope to do well in an absolute sense. However, every day, before you make your prediction, you get to hear the advice of a bunch of experts, who make their own predictions. These "experts" may or may not know what they are talking about, and you start off knowing nothing about them. Nevertheless, you want to come up with a rule to aggregate their advice so that you end up doing (almost) as well as the best expert (whomever he might turn out to be) in hindsight. Sounds tough.

Lets start with an even easier case:

- There are N experts who will make predictions in T rounds.
- At each round t, each expert i makes a prediction $p_i^t \in \{U, D\}$ (up or down).
- We (the algorithm) aggregate these predictions somehow, to make our own prediction $p_A^t \in \{U, D\}$. Then we learn the true outcome $o^t \in \{U, D\}$. If we predicted incorrectly (i.e. $p_A^t \neq o^t$), then we *made a mistake.*
- To make things easy, we will assume at first that there is one *perfect* expert who never makes a mistake (but we don't know who he is).

Can we find a strategy that is guaranteed to make at most $\log(N)$ mistakes?

We can, using the simple halving algorithm!

Algorithm 1 The Halving Algorithm

```
Let $S^1 \leftarrow \{1, \ldots, N\}$ be the set of all experts.
for $t = 1$ to $T$ do
    Let $S_U^t = \{i \in S : p_i^t = U\}$ be the set of experts in $S^t$ who predict up, and $S_D^t = S^t \setminus S_U^t$ be the set
    who predict down.
    Predict with the majority vote: If $|S_U^t| > |S_D^t|$, predict $p_A^t = U$, else predict $p_A^t = D$.
    Eliminate all experts that made a mistake: If $o^T = U$, then let $S^{t+1} = S_U^t$, else let $S^{t+1} = S_D^t$
end for
```

Its not hard to see that the halving algorithm makes at most $\log N$ mistakes under the assumption that one expert is perfect:

Theorem 1 *If there is at least one perfect expert, the halving algorithm makes at most* $\log N$ *mistakes.*

Proof Since the algorithm predicts with the majority vote, every time it makes a mistake at some round t, at least half of the remaining experts have made a mistake and are eliminated, and hence $|S^{t+1}| \leq |S^t|/2$. On the other hand, the perfect expert is never eliminated, and hence $|S^t| \geq 1$ for all t. Since $|S^1| = N$, this means there can be at most $\log N$ mistakes. ■

Not bad – $\log N$ is pretty small even if N is large (e.g. if $N = 1024$, $\log N = 10$, if $N = 1,048,576$, $\log N = 20$), and doesn't grow with T, so even with a huge number of experts, the average number of mistakes made by this algorithm is tiny.

What if no expert is perfect? Suppose the best expert makes OPT mistakes. Can we find a way to make not too many more than OPT mistakes?

The first approach you might try is the iterated halving algorithm:

Algorithm 2 The Iterated Halving Algorithm

```
Let $S^1 \leftarrow \{1, \dots, N\}$ be the set of all experts.
for $t = 1$ to $T$ do
  If $|S^t| = 0$ Reset: Set $S^t \leftarrow \{1, \dots, N\}$.
  Let $S_U^t = \{i \in S : p_i^t = U\}$ be the set of experts in $S^t$ who predict up, and $S_D^t = S^t \setminus S_U^t$ be the set who predict down.
  Predict with the majority vote: If $|S_U^t| > |S_D^t|$, predict $p_A^t = U$, else predict $p_A^t = D$.
  Eliminate all experts that made a mistake: If $o^T = U$, then let $S^{t+1} = S_U^t$, else let $S^{t+1} = S_D^t$
end for
```

Theorem 2 *The iterated halving algorithm makes at most* $\log(N)(OPT + 1)$ *mistakes.*

Proof As before, whenever the algorithm makes a mistake, we eliminate half of the experts, and so the algorithm can make at most $\log N$ mistakes between any two resets. But if we reset, it is because since the last reset, *every* expert has made a mistake: in particular, between any two resets, the *best* expert has made at least 1 mistake. This gives the claimed bound. ■

We should be able to do better though. The above algorithm is wasteful in that every time we reset, we forget what we have learned! The weighted majority algorithm can be viewed as a softer version of the halving algorithm: rather than eliminating experts who make mistakes, we just down-weight them:

Algorithm 3 The Weighted Majority Algorithm

```
Set weights $w_i^1 \leftarrow 1$ for all experts $i$.
for $t = 1$ to $T$ do
  Let $W_U^t = \sum_{i:p_i^t=U} w_i$ be the weight of experts who predict up, and $W_D^t = \sum_{i:p_i^t=D} w_i$ be the weight of those who predict down.
  Predict with the weighted majority vote: If $W_U^t > W_D^t$, predict $p_A^t = U$, else predict $p_A^t = D$.
  Down-weight experts who made mistakes: For all $i$ such that $p_i^t \neq o^t$, set $w_i^{t+1} \leftarrow w_i^t/2$
end for
```

Theorem 3 *The weighted majority algorithm makes at most* $2.4\,(OPT + \log(N))$ *mistakes.*

Note that $\log(N)$ is a fixed constant, so the ratio of mistakes the algorithm makes compared to OPT is just 2.4 in the limit – not great, but not bad.

Proof Let M be the total number of mistakes that the algorithm makes, and let $W^t = \sum_i w_i^t$ be the total weight at step t. Note that on any round t in which the algorithm makes a mistake, at least half of the total weight (corresponding to experts who made mistakes) is cut in half, and so $W^{t+1} \leq (3/4)W^t$.

Hence, we know that if the algorithm makes M mistakes, we have $W^T \leq N \cdot (3/4)^M$. Let i^* be the best expert. We also know that $w_i^T = (1/2)^{\text{OPT}}$, and so in particular, $W^T > (1/2)^{\text{OPT}}$. Combining these two observations we know:

$$\left(\frac{1}{2}\right)^{\text{OPT}} \leq W \leq N \left(\frac{3}{4}\right)^M$$

$$\left(\frac{4}{3}\right)^M \leq N \cdot 2^{\text{OPT}}$$

$$M \leq 2.4(\text{OPT} + \log(N))$$

as claimed. ■

We've been doing well; lets get greedy. What do we want in an algorithm? We might want:

1. It to make only 1 times as many mistakes as the best expert in the limit, rather than 2.4 times...
2. It to be able to handle N distinct actions (a separate action for each expert), not just two (up and down)...
3. It to be able to handle experts having arbitrary costs in $[0, 1]$ at each round, not just binary costs (right vs. wrong)

Formally, we want an algorithm that works in the following framework:

1. In rounds $1, \ldots, T$, the algorithm chooses some expert i^t.
2. Each expert i experiences a loss $\ell_i^t \in [0, 1]$. The algorithm experiences the loss of the expert it chooses: $\ell_A^t = \ell_{i^t}^t$.
3. The total loss of expert i is $L_i^T = \sum_{t=1}^T \ell_i^t$, and the total loss of the algorithm is $L_A^T = \sum_{t=1}^T \ell_A^t$. The goal of the algorithm is to obtain loss not much worse than that of the best expert: $\min_i L_i^T$.

The polynomial weights algorithm can be viewed as a further smoothed version of the weighted majority algorithm, and has a parameter ϵ which controls how quickly it down-weights experts. Notably, it is *randomized*: rather than making deterministic decisions, it randomly chooses an expert to follow with probability proportional to their weight.

Algorithm 4 The Polynomial Weights Algorithm (PW)

```
Set weights $w_i^1 \leftarrow 1$ for all experts $i$.
for $t = 1$ to $T$ do
    Let $W^t = \sum_{i=1}^N w_i^t$.
    Choose expert $i$ with probability $w_i^t / W^t$.
    For each $i$, set $w_i^{t+1} \leftarrow w_i^t \cdot (1 - \epsilon \ell_i^t)$.
end for
```

Theorem 4 *For any sequence of losses, and any expert k:*

$$\frac{1}{T} \mathrm{E}[L_{PW}^T] \leq \frac{1}{T} L_k^T + \epsilon + \frac{\ln(N)}{\epsilon \cdot T}$$

In particular, setting $\epsilon = \sqrt{\frac{\ln(N)}{T}}$ we get:

$$\frac{1}{T} \mathrm{E}[L_{PW}^T] \leq \frac{1}{T} \min_k L_k^T + 2\sqrt{\frac{\ln(N)}{T}}$$

In other words, the average loss of the algorithm quickly approaches the average loss of the best expert exactly, at a rate of $1/\sqrt{T}$. Note that this works against an *arbitrary* sequence of losses, which might be chosen adaptively by an adversary. This is pretty incredible. And it will be the source of the power of this framework in applications: we (the algorithm designer) can play the role of the adversary to get the results that we want.

Ok, on to the proof:

Proof Let F^t denote the expected loss of the polynomial weights algorithm at time t. By linearity of expectation, we have $\mathrm{E}[L_{PW}^T] = \sum_{t=1}^T F^t$. We also know that:

$$F^t = \frac{\sum_{i=1}^N w_i^t \ell_i^t}{W^t}$$

How does W^t change between rounds? We know that $W^1 = N$, and looking at the algorithm we see:

$$W^{t+1} = W^t - \sum_{i=1}^N \epsilon w_i^t \ell_i^t = W^t(1 - \epsilon F^t)$$

So by induction, we can write:

$$W^{T+1} = N \prod_{t=1}^T (1 - \epsilon F^t)$$

Taking the log, and using the fact that $\ln(1-x) \leq -x$, we can write:

$$\begin{aligned} \ln(W^{t+1}) &= \ln(N) + \sum_{t=1}^T \ln(1 - \epsilon F^t) \\ &\leq \ln(N) - \epsilon \sum_{t=1}^T F^t \\ &= \ln(N) - \epsilon \mathrm{E}[L_{PW}^T] \end{aligned}$$

Similarly (using the fact that $\ln(1-x) \geq -x - x^2$ for $0 < x < \frac{1}{2}$), we know that for every expert k:

$$\begin{aligned} \ln(W^{T+1}) &\geq \ln(w_k^{T+1}) \\ &= \sum_{t=1}^T \ln(1 - \epsilon \ell_k^t) \\ &\geq -\sum_{t=1}^T \epsilon \ell_k^t - \sum_{t=1}^T (\epsilon \ell_k^t)^2 \\ &\geq -\epsilon L_k^T - \epsilon^2 T \end{aligned}$$

Combining these two bounds, we get:

$$\ln(N) - \epsilon L_{PW}^T \geq -\epsilon L_k^T - \epsilon^2 T$$

for all k. Dividing by ϵ and rearranging, we get:

$$L_{PW}^T \leq \min_k L_k^T + \epsilon T + \frac{\ln(N)}{\epsilon}$$

■

One final observation: we have described the algorithm so far as if it is *randomly* selecting some action i at each round, and have been measuring its expected loss at each round: $\sum_{i=1}^{N} \frac{w_i^t}{W^t}\ell_i^t$. This makes sense if the algorithm must choose an expert to play at every round. But in some settings, it makes sense for the algorithm to play a *vector* in $\Delta[n] = \{p \in [0,1]^N : \sum_{i=1}^{n} p_i = 1\}$ at every round. For example, it might be interacting in the following setting, called online adversarial linear optimization:

1. In rounds $1, \ldots, T$ the algorithm chooses a vector $w^t \in \Delta[N]$.
2. The adversary chooses a loss vector $\ell^t \in [0,1]^N$.
3. The algorithm experiences loss $\ell_A^t = \langle w^t, \ell^t \rangle$.
4. The goal of the algorithm is to guarantee that:

$$\frac{1}{T}\sum_{t=1}^{T}\langle w^t, \ell^t\rangle \leq \min_{w^* \in \Delta[n]} \frac{1}{T}\sum_{t=1}^{T}\langle w^*, \ell^t\rangle + o(1).$$

In this case, we can view the exact same algorithm we have derived and analyzed as a deterministic algorithm for choosing such a vector — at round t is plays the vector $w^t = \{\frac{w_i^t}{W^t}\}_{i=1}^n$.

Algorithm 5 The Polynomial Weights Algorithm for Online Linear Optimization

```
Set weights $w_i^1 \leftarrow 1$ for all experts $i$.
for $t = 1$ to $T$ do
  Let $W^t = \sum_{i=1}^{N} w_i^t$.
  Play vector $w^t = \{\frac{w_i^t}{W^t}\}_{i=1}^n$
  For each $i$, set $w_i^{t+1} \leftarrow w_i^t \cdot (1 - \epsilon \ell_i^t)$.
end for
```

Our existing analysis proves the following theorem:

Theorem 5 *Setting* $\epsilon = \sqrt{\frac{\ln(N)}{T}}$*, for any sequence of losses* $\ell^t \in [0,1]^N$*:*

$$\frac{1}{T}\sum_{t=1}^{T}\langle w^t, \ell^t\rangle \leq \min_{w^* \in \Delta[n]} \frac{1}{T}\sum_{t=1}^{T}\langle w^*, \ell^t\rangle + 2\sqrt{\frac{\ln(N)}{T}}$$

Proof The left hand side is exactly the expected loss of the polynomial weights algorithm we analyzed in the experts setting. Continuing the translation, the "loss of expert i" corresponds to $\frac{1}{T}\sum_{t=1}^{T}\langle e_i, \ell^t\rangle$, where e_i is the i'th standard basis vector (with a 1 in the i'th coordinate and a 0 in every other coordinate). Finally observe that we always have that for every sequence of losses:

$$\min_{w^* \in \Delta[n]} \sum_{t=1}^{T}\langle w^*, \ell^t\rangle = \sum_{t=1}^{T}\langle e_{i^*}, \ell^t\rangle$$

where $i^* = \arg\min_{i \in [N]} \sum_{t=1}^{T} \ell_i^t$. Hence regret to the best basis vector e_{i^*} (i.e. the best expert) implies regret to the best vector $w^* \in \Delta[N]$. ■

Finally, we observe that there if we are using polynomial weights for online linear optimization, there is no reason to restrict attention to vectors w^* whose coordinates sum to 1, or losses that lie in the range $[0,1]$. We simply have to pay for the scale of the vectors we are optimizing over. Lets see how we could use the polynomial weights algorithm to solve the online linear optimization problem over the set of non-negative vectors w that sum to at most R_1: $B_N(R_1) = \{w \in \mathbb{R}^N_{\geq 0} : \sum_{i=1}^n \leq R_1\}$, for loss functions that take values in the range $\ell_i^t \in [-R_2/2, R_2/2]$.

First lets deal with the issue of having coordinates of w that sum to *at most* some value R_1 rather than exactly R_1. We can simply add an extra $N+1$'st coordinate that always has loss $\ell^t_{N+1} = 0$. Running our algorithm in this augmented $N+1$ dimensional space means that if the $N+1$ dimensional vector w^t has coordinates summing to exactly R_1 at every round, the first N coordinates of w (the "real ones") sum to at most R_1 — and the algorithm experiences the same loss as if it played in only the real N dimensional space.

Next lets deal with the issue of negative losses. This is also easy: simply shift them by adding $R_2/2$ to every coordinate. Now we have $\ell_i^t \in [0, R_2]$, and note that the regret to any target w^* remains unchanged under this shift, because:

$$\langle w^t, \ell^t + R_2/2\rangle - \langle w^*, \ell^t + R_2/2\rangle = \big(\langle w^t, \ell^t\rangle - \langle w^*, \ell^t\rangle\big) + \big(\langle R_2/2, \ell^t\rangle - \langle R_2/2, \ell^t\rangle\big) = \langle w^t, \ell^t\rangle - \langle w^*, \ell^t\rangle$$

So regret bounds for the shifted space hold also for the original losses.

We're almost done. We simply have to scale down everything, apply our bounds, and then remember to scale back up. Suppose we divide the coordinates of w^t by R_1 and the coordinates of ℓ^t by R_2. We are now in the setting for which we have proven the regret bound for the polynomial weights algorithm, and so we have that the polynomial weights algorithm can obtain the regret bound:

$$\frac{1}{T}\sum_{t=1}^T \langle \frac{w^t}{R_1}, \frac{\ell^t}{R_2}\rangle \leq \min_{w^* \in \Delta[n]} \frac{1}{T}\sum_{t=1}^T \langle \frac{w^*}{R_1}, \frac{\ell^t}{R_2}\rangle + 2\sqrt{\frac{\ln(N)}{T}}$$

Multiplying this bound through by $R_1 \cdot R_2$ we obtain:

Theorem 6 *For any sequence of losses $\ell^t \in [-R_2/2, R_2/2]^N$, the polynomial weights algorithm can be used to play vectors $w^t \in B_N(R_1)$ and obtain:*

$$\frac{1}{T}\sum_{t=1}^T \langle w^t, \ell^t\rangle \leq \min_{w^* \in B_N(R_1)} \frac{1}{T}\sum_{t=1}^T \langle w^*, \ell^t\rangle + 2R_1R_2\sqrt{\frac{\ln(N)}{T}}$$

CIS 320: Algorithms November 3, 2021

Lecture 17

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Linear Programs

One of the most powerful optimization paradigms that have worst-case efficient solvers are so-called "Linear Programs".

A linear program is defined by a set of d variables to optimize over, a linear function of those variables to optimize, and constraints on how we can set the variables, specified as linear inequalities.

Definition 1 *A linear program is an optimization problem defined over n non-negative decision variables $x_1, \ldots, x_n \geq 0$, a linear objective function, and d linear constraints. It takes the form:*

$$\textit{Maximize} \sum_{i=1}^{n} c_i x_i$$

$$\textit{such that for each constraint } j \in [d]:$$

$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_i$$

We start by observing that linear programs are somewhat more general than they appear at first glance. For example, because the constants $a_{i.j}, b_i$ can be negative, they can also encode inequality constraints in the other direction: If we want to express the inequality $\sum_{i=1}^{n} a_{i,j} x_i \geq b_i$, we can write it as $\sum_{i=1}^{n} -a_{i,j} x_i \leq -b_i$ instead. Similarly, if we want to minimize some objective rather than maximize it, we can express that by multiplying the coefficients c_i by -1. Finally, we can express equality constraints as pairs of inequality constraints: $\sum_{i=1}^{n} a_{i,j} = b$ can be represented by two constraints, $\sum_{i=1}^{n} a_{i,j} \leq b$ and $\sum_{i=1}^{n} a_{i,j} \geq b$

Lots of things can be represented as linear programs. Classically, linear programs were used to express production problems. For example:

> A lumber company can produce either pallets or high quality lumber. It cannot produce more than 200 units (thousand board feet) of lumber per day, which maxes out usage of their kiln, and it cannot produce more than 600 pallets per day. Its main saw can process at most 400 logs per day. 1 unit of lumber requires 1.4 logs, and one pallet requires 0.25 logs. High quality logs used for lumber cost \$200 per log, and low quality logs used for pallets cost \$4 per log. Processing lumber costs \$200 per unit, and processing pallets costs \$5. A unit of lumber sells for \$490 per unit, and a pallet sells for \$9. How many pallets and units of lumber should the lumber company produce?

We can directly represent this as a linear program Say that x_L represents the units of lumber to produce, x_P represents the number of pallets, y_H represents the number of high quality logs purchased, and y_L represents the number of low quality logs. Then the problem is to solve:

$$\text{Maximize } 290 \cdot x_L + 4 \cdot x_P - 200 y_H - 4 \cdot y_L$$

such that:

$$x_L \leq 200 \quad x_P \leq 600 \quad 1.4 \cdot x_L \leq y_H \quad 0.25 x_P \leq y_L \quad y_L + y_H \leq 400$$

But observe that we can also write the max-flow problem as a linear program. Suppose we have a flow network $C = (V, E)$ with costs c_e (recall without loss of generality we assume there are no incoming edges to e and no outgoing edges from t). The max flow problem can be expressed as:

$$\text{Maximize} \sum_{e \text{ out of } s} f(e)$$

such that:

$$\text{For every } e \in E : f(e) \leq c_e \quad \text{and} \quad \text{for every } v \notin \{s,t\} : \sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

We could also write down linear programs for the bipartite matching, min-cut, minimum spanning tree, and other problems we've studied in this course (although sometimes some cleverness would be needed to argue that they have integer optimal solutions).

It turns out that we can solve linear programs efficiently and in time polynomial in the number of variables and constraints! The story is a little complicated — some of the most efficient algorithms in practice (Simplex) are not polynomial time in the worst case, and some of the polynomial time algorithms (Ellipsoid) are not efficient in practice. In this lecture, we'll show how to use the polynomial weights algorithm we derived last lecture to give approximate solutions to linear programs. A benefit of this approach is that we never need to enumerate all of the constraints, we only need to find violated constraints when they exist. So this lets us efficiently approximate the solutions to linear programs with exponentially many constraints, so long as we can efficiently identify violated constraints given a candidate solution.

To do so, we'll first convert a linear program into a linear feasibility problem, which is just a linear program without the objective

Definition 2 *A linear feasibility problem is defined over n non-negative decision variables $x_1, \ldots, x_n \geq 0$ and d linear constraints. It is the problem of finding values for the x_i such that for each constraint $j \in [d]$:*

$$\sum_{i=1}^{n} a_{i,j} x_i \leq b_i$$

We first observe that if a linear program has a solution x with optimal objective value OPT, then we can write it as a linear feasibility problem simply by adding the constraint that the objective take its optimal value:

$$\sum_{i=1}^{n} -c_i x_i \leq -\text{OPT}$$

Of course we don't know OPT, but if we had the ability to solve linear feasibility problems, then we could find it via binary search. So from here on out, we'll focus on solving linear feasibility problems.

First let us recall the final guarantee we derived last lecture for using the polynomial weights algorithm for online linear optimization:

Theorem 3 *For any sequence of losses $\ell^t \in [-R_2/2, R_2/2]^N$, the polynomial weights algorithm can be used to play vectors $w^t \in B_N(R_1)$ and obtain:*

$$\frac{1}{T}\sum_{t=1}^{T} \langle w^t, \ell^t \rangle \leq \min_{w^* \in B_N(R_1)} \frac{1}{T}\sum_{t=1}^{T} \langle w^*, \ell^t \rangle + 2R_1 R_2 \sqrt{\frac{\ln(N)}{T}}$$

Our goal is to leverage this theorem to solve linear programs. Our plan will be to run the polynomial weights algorithm, which maintains a vector w^t that we will treat as a candidate solution x to our linear feasibility problem. At every round, we will check whether it (approximately) satisfies all of the constraints. If it does, we're done, and we'll return the solution $x = w^t$. Otherwise, we'll run the polynomial weights algorithm for another round, by feeding it a loss vector ℓ^{t+1} defined by one of the constraints that is violated. The algorithm is as follows:

Algorithm 1 Solve($\{a,b\}_{j=1}^d, R_1, R_2$)

```
**Initialize** the polynomial weights algorithm, parameterized to produce vectors $w \in B_d(R_1)$ and receive losses in $[-R_2/2, R_2/2]$.
**Let** $t = 1$, and $w^1 \in \mathbb{R}^n$ be the vector representing the state of the PW algorithm.
**while** There exists a constraint $j^t$ such that $\sum_{i=1}^n a_{i,j^t} w_i^t \geq b_i + \alpha$ **do**
    **Run** the PW algorithm for another iterate using loss function $\ell^t \in \mathbb{R}^n$ defined so that $\ell_i^t = a_{i,j^t}$.
    **Let** $t \leftarrow t+1$ and $w^t$ be the updated state of the PW algorithm.
**end while**
Output $x = w^t$.
```

Note that we need to pass to this algorithm an upper bound R_1 on the scale of a feasible solution, and an upper bound R_2 on the quantities $a_{i,j}$, but we can do this — we'll quickly work out how to do it for the max flow problem at the end. It will not exactly solve feasibility problems, but rather will return α-approximate solutions:

Definition 4 *Given a linear feasibility problem $\{a,b\}_{j=1}^d$, x is an α-feasible solution if for all constraints j, we have:*

$$\sum_{i=1}^n x_i a_{i,j} \leq b_i + \alpha$$

The analysis turns out to be very direct and simple (which is to say, we already did most of the work when we analyzed the polynomial weights algorithm):

Theorem 5 *Let $\{a,b\}_{j=1}^d$ be a linear feasibility problem that has a feasible solution $x^* \in B_n(R_1)$, and such that* $\max |a_{i,j}| \leq R_2/2$. *Then Solve($\{a,b\}_{j=1}^d, R_1, R_2$) returns an α-feasible solution after at most*

$$T \leq \frac{4R_1^2 R_2^2 \ln(n)}{\alpha^2}$$

many iterations.

Proof First, observe that by construction, if the algorithm returns a solution $x = w^t$ for some t, it is an α-feasible solution, so it only remains to argue that the algorithm halts and returns something after at most T iterations. By assumption, there exists $x^* \in B_n(R_1)$ such that for every constraint j, $\sum_{i=1}^n x_i^* a_{i,j} \leq b_j$. We will consider the polynomial weights algorithm regret to x_i^* and derive a contradiction if the algorithm has not returned a solution after T iterations.

By construction, the loss function ℓ^t is defined so that at every round, $\ell^t = a_{j^t}$ for some constraint j^t. On the one hand, we know that $\sum_{i=1}^n x_i^* \ell_i^t \leq b_{j^t}$ by the feasibility of x^*. On the other hand, we know that by construction, $\sum_{i=1}^n w_i^t \ell_i^t \geq b_{j^t} + \alpha$ by definition of the algorithm. Hence the regret of the polynomial weights algorithm is at least:

$$\frac{1}{T}\sum_{t=1}^T \langle w^t, \ell^t \rangle - \frac{1}{T}\sum_{t=1}^T \langle x^*, \ell^t \rangle \geq \frac{1}{T}\sum_{t=1}^T (b_{j^t} + \alpha - b_{j^t}) \geq \alpha$$

On the other hand, the regret bound of the polynomial weights algorithm implies:

$$\alpha \leq \frac{1}{T}\sum_{t=1}^T \langle w^t, \ell^t \rangle - \frac{1}{T}\sum_{t=1}^T \langle x^*, \ell^t \rangle \leq 2R_1 R_2 \sqrt{\frac{\ln(n)}{T}}$$

Hence it must be that:

$$2R_1 R_2 \sqrt{\frac{\ln(n)}{T}} \geq \alpha$$

Solving for T gives the theorem. ■

Lets briefly return to the max-flow problem. How can we bound R_1 the norm of a feasible solution? We can upper bound this by observing that a feasible flow in the worst case saturates the capacity of every single edge, and so we have $R_1 \leq C \equiv \sum_{e \in E} c_e$. Note that this is the same quantity C that we used to bound the running time of the specialized algorithm we derived for max-flow. By inspection, we have that $\max_{i,j} |a_{i.j}| = 1$, and so we can take $R_2 = 2$.

CIS 320: Algorithms November 8, 2021

Lecture 18

Lecturer: Aaron Roth *Scribe: Aaron Roth*

The Minimax Theorem and Equilibria of Zero Sum Games

In this lecture we'll take a brief interlude from linear programs — we'll get back to them in a couple of lectures. We will prove the fundamental *minimax* theorem for zero sum games, which turns out to be closely related to *linear programming duality.* We'll also show how to actually compute equilibria of zero sum games. Remarkably, all of this will follow once again from our analysis of the polynomial weights algorithm.

First, what is a "zero-sum game"? It is a model of a strictly adversarial interaction, in which one player's sole objective is to minimize some cost function, and their opponent's sole objective is to maximize it. The players have the ability to choose amongst a set of actions, which jointly determine the cost. One can define this somewhat more generally, but it will suffice for us to talk about players with finite sets of actions who are allowed to choose probability distributions over those actions (i.e. to randomize)

Definition 1 *A* zero sum game *is defined by an action set* $A_1 = \{1, \ldots, m\}$ *for the* minimization player*, an action set* $A_2 = \{1, \ldots, n\}$ *for the* maximization player*, and a cost function* $C : A_1 \times A_2 \to \mathbb{R}$*.*

We can represent a zero sum game by thinking of C as a matrix, in which the rows correspond to actions of the minimization player (lets call her Min), columns correspond to actions of the maximization player (lets call him Max), and the entries record the costs that result from the corresponding choices of actions by Min and Max. For example, you might recognize the following zero sum game as "Rock Paper Scissors":

	Rock	Paper	Scissors
Rock	1	2	0
Paper	0	1	2
Scissors	2	0	1

A (*mixed*) strategy for Min corresponds to a probability distribution over her actions: $p \in \Delta[m]$. Similarly, a mixed strategy for Max corresponds to a probability distribution over his actions: $q \in \Delta[n]$. When players randomize, we compute the *expected* cost of the resulting outcome:

$$C(p, q) = \sum_{i=1}^{m} \sum_{j=1}^{n} C[i, j] p_i q_j = q^T C p$$

If one of the players plays a *pure* strategy — i.e. does not randomize — for example, Max might deterministically play $y \in [n]$ we will abuse notation and write:

$$C(p, y) = \sum_{i=1}^{m} C[i, y] p_i = e_y^T C p$$

Normally, Rock Paper Scissors is played as a simultaneous move game. But what if Min were forced to play at a disadvantage, by having to first *announce* her strategy to Max, who would then get to best respond? If she played the strategy $p = (2/3, 1/3, 1/3)$, Max would exploit the fact that Min was playing Rock too frequently, and play paper in response, resulting in cost $2/3 \cdot 2 + 1/3 \cdot 1 + 1/3 \cdot 0 = 5/3$. Instead, she should play so as to minimize the cost that results *after* Max best response. Similarly, if Max is handicapped by the need to go first and announce his strategy before Min gets an opportunity to best respond, what he should do is play so as to maximize the cost *after* Min best responds by choosing the action with minimum cost. We can define the corresponding MinMax and MaxMin values of the game:

Definition 2 *For an $n \times m$ matrix C:*

$$\max\min(C) = \max_{q \in \Delta[n]} \min_{x \in [m]} \sum_{j=1}^{n} q_i \cdot C[x, i]$$

$$\min\max(C) = \min_{p \in \Delta[m]} \max_{y \in [n]} \sum_{i=1}^{m} p_i \cdot C[i, y]$$

Here note that we have the player who goes second playing a single action, rather than a distribution over actions — but this is without loss of generality, since a player's *best response* need never be randomized. (because an average over a bunch of numbers can never be smaller than the minimum or larger than the maximum) Of course, in Rock Paper Scissors it doesn't matter who goes first: either player will randomize uniformly across Rock Paper and Scissors if they go first, which will make their opponent indifferent between their options: optimal play obtains cost $1/3 \cdot 1 + 1/3 \cdot 2 + 1/3 \cdot 0 = 1$ in both cases.

This turns out to be more generally true in zero-sum games: it doesn't matter who goes first. This is a surprisingly deep (and useful) fact known as Von Neumann's minimax theorem. For Min, going first is clearly only a disadvantage, since she is revealing information to Max, and so we know that $\min\max(C) \geq \max\min(C)$. The minimax theorem says that this inequality is in fact an equality:

Theorem 3 (Von Neumann) *In any zero sum game C:*

$$\min\max(C) = \max\min(C)$$

The theorem is not obvious... Von Neumann proved it in 1928, and said "As far as I can see, there could be no theory of games ... without that theorem ... I thought there was nothing worth publishing until the Minimax Theorem was proved". Previously, Borell had proven it for the special case of 5×5 matrices, and thought it was false for larger matrices.

However. Now that we know of the polynomial weights algorithm, we can provide a very simple, constructive proof. In fact, what we'll do is give an algorithm that explicitely constructs strategies p, q for Min and Max respectively such that (p, q) form an ϵ-approximate *minimax equilibrium.* Von Neumann's minimax theorem will follow as a corollary.

Definition 4 *Vectors $p \in \Delta[m]$, $q \in \Delta[n]$ form an ϵ-approximate minimax equilibrium with respect to a game C if:*

$$\max_{y \in [n]} C(p, y) - \epsilon \leq C(p, q) \leq \min_{x \in [m]} C(x, q) + \epsilon$$

Algorithm 1 ComputeEQ(C, ϵ)

```
Let $T \leftarrow \frac{4 \log m}{\epsilon^2}$
Initialize a copy of polynomial weights to run over $w^t \in \Delta^m$.
for $t = 1$ to $T$ do
    Let $y^t = \arg\max_{y \in [n]} C(w^t, y)$
    Let $\ell^t \in [0, 1]^m$ be such that $\ell^t_i = C[i, y_t]$.
    Pass $\ell^t$ to the PW algorithm.
end for
Let $\bar{x} = \frac{1}{T} \sum_{t=1}^{T} w^t$ and $\bar{y} = \frac{1}{T} \sum_{t=1}^{T} e_{y^t}$.
Return $(\bar{x}, \bar{y})$.
```

In the algorithm above, $e_i \in [0, 1]^n$ refers to the i'th standard basis vector — i.e. it has a 1 in its i'th index and a 0 in every other index.

Theorem 5 *For any $\epsilon > 0$, and any $n \times m$ 0-sum game C, ComputeEQ(C, ϵ) returns vectors $(\bar{x}, \bar{y})$ that form an ϵ-approximate minimax equilibrium.*

To see that Von Neumann's theorem follows as a corollary, note that this implies that for any $\epsilon > 0$, we can find $\bar{x}, \bar{y}$ such that:

$$\min\max(C) - \epsilon \leq \max_{y \in [n]} C(\bar{x}, y) - \epsilon \leq C(\bar{x}, \bar{y}) \leq \min_{x \in [m]} C(x, \bar{y}) + \epsilon \leq \max\min(C) + \epsilon$$

Thus we have $\min\max(C) \leq \max\min(C) + 2\epsilon$ for every ϵ, so it must be that $\min\max(C) = \max\min(C)$.

Now to prove the theorem:

Proof We begin with a useful observation coming from linearity. Let $x^* \in [m]$ be any fixed action. Then:

$$\begin{aligned} \frac{1}{T}\sum_{t=1}^{T} C(x^*, y^t) &= C\left(x^*, \frac{1}{T}\sum_{t=1}^{T} e_{y^t}\right) \\ &= C(x^*, \bar{y}) \end{aligned}$$

Similarly, for any fixed y^* $\frac{1}{T}C(w^t, y^*) = C(\bar{x}, y^*)$. Now suppose Min and Max are playing using $\bar{x}$ and $\bar{y}$ respectively. Let $x^* = \arg\min_x C(x, \bar{y})$ and $y^* = \arg\max_y C(\bar{x}, y)$ be their best responses. By definition:

$$C(x^*, \bar{y}) \leq C(\bar{x}, \bar{y}) \leq C(\bar{x}, y^*)$$

We also know that by the guarantee of the polynomial weights algorithm that on the one hand:

$$\begin{aligned} \frac{1}{T}\sum_{t=1}^{T} C(w^t, y^t) &\leq \frac{1}{T}\sum_{t=1}^{T} C(x^*, y^t) + \sqrt{\frac{4\log m}{T}} \\ &= C(x^*, \bar{y}) + \sqrt{\frac{4\log m}{T}} \end{aligned}$$

And on the other hand, by definition of the choice of the y^t:

$$\begin{aligned} \frac{1}{T}\sum_{t=1}^{T} C(w^t, y^t) &\geq \frac{1}{T}\sum_{t=1}^{T} C(w^t, y^*) \\ &= C(\bar{x}, y^*) \end{aligned}$$

Subtracting the second inequality from the first, we have:

$$0 \leq C(x^*, \bar{y}) - C(\bar{x}, y^*) + \sqrt{\frac{4\log m}{T}}.$$

Adding and subtracting $C(\bar{x}, \bar{y})$ and multiplying by -1 we get:

$$(C(\bar{x}, \bar{y}) - C(x^*, \bar{y})) + (C(\bar{x}, y^*) - C(\bar{x}, \bar{y})) \leq \sqrt{\frac{4\log m}{T}}$$

Finally recall that by definition of x^* and y^*, we have that both terms on the left hand side are non-negative. Thus we have that individually:

$$(C(\bar{x}, \bar{y}) - C(x^*, \bar{y})) \leq \sqrt{\frac{4\log m}{T}} \quad (C(\bar{x}, y^*) - C(\bar{x}, \bar{y})) \leq \sqrt{\frac{4\log m}{T}}$$

which implies that $(\bar{x}, \bar{y})$ form a $\sqrt{\frac{4\log m}{T}}$-approximate minimax equilibrium. By our choice of T, $\sqrt{\frac{4\log m}{T}} = \epsilon$. ■

CIS 320: Algorithms November 10, 2021

Lecture 19

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Boosting

In this class we'll derive a powerful family of machine learning algorithms, taking advantage of the polynomial weights algorithm and the minimax theorem we proved last class. First some basic definitions:

Definition 1 *A labeled* datapoint *is a pair $(x, y) \in X \times Y$, where X is some space of* features *and Y is some space of* labels*: for example, a common case is $X = \mathbb{R}^d$, and $Y = \{0, 1\}$.*
A dataset $D \in (X \times Y)^n$ is a collection of n labeled datapoints.

Our goal will be to find some function $f : X \to Y$ for predicting labels from their features that has high accuracy:

Definition 2 *Given a predictor $f : X \to Y$, its prediction accuracy on a dataset D is:*

$$acc(f, D) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}[f(x_i) = y_i]$$

The prediction accuracy as defined uniformly weights all of the points in the dataset. But we can also define weighted *prediction accuracy relative to any other weighting $w \in \Delta[n]$ of the n points:*

$$acc(f, D, w) = \sum_{i=1}^{n} w_i \mathbb{1}[f(x_i) = y_i]$$

Note that $acc(f, D)$ is simply the special case of $acc(f, D)$ in which $w_i = 1/n$ for all i.

Remark We're ignoring an important statistical aspect of machine learning here: our goal is typically not actually to predict the labels of the points in our dataset D, but to predict the labels of new points drawn from the same distribution as D that we have never seen before. Doing so requires making high accuracy predictions using "simple" hypotheses. We'll focus on the algorithmic aspect here, but the "boosting" approach we discuss here *also* has good statistical generalization properties.

Definition 3 *A hypothesis class H is a collection of predictors or* hypotheses *$h : X \to Y$. A weighted learning algorithm A with range H is a mapping from datasets and weight vectors to hypotheses in H. $A : (X \times Y)^n \times [0, 1]^n \to H$.*

If (e.g.) $Y = \{0, 1\}$ then it is generally uninteresting to find a hypothesis h that has $acc(h, D) \leq 1/2$, since we could obtain that just by randomly guessing or always predicting the most common label. But what about if we could come up with a hypothesis that has just *slightly* better accuracy: $acc(h, D) = 0.51$. Would that be interesting? This is often not hard: it might involve simply finding a single feature that has a small correlation with the label. We'll define a weak learning algorithm for a dataset D as one that can always come up with a hypothesis with weighted accuracy better than random guessing, for any weight vector:

Definition 4 *A weighted learning algorithm A is a weak learning algorithm for D if for every distribution $w \in \Delta[n]$, $A(D, w) = h$ such that:*

$$acc(h, D, w) \geq 0.51$$

Remark If the notion of a "weighted" learning algorithm seems odd, observe that one way to implement a weighted learning algorithm is to construct a new dataset D' by sampling from D under the probability distribution specified by w, and then running a regular old (unweighted) learning algorithm on D'. We'll use weighted learning algorithms just because it avoids the need to argue about subsampling.

A weak learning algorithm seems — well, weak. It only guarantees to beat random guessing by a tiny amount. But remarkably, we will show that if there is a computationally efficient weak learning algorithm for D, then there is also a computationally efficient strong learning algorithm for D — one that can find a perfect predictor.

Definition 5 *A is a strong learning algorithm for D if $A(D) = h$ such that $acc(h, D) = 1$.*

Theorem 6 *For any dataset D, if there exists an efficient (polynomial time) weak learning algorithm A for D, then there exists an efficient strong learning algorithm A' for D.*

Proof Our construction will involve applying the minimax theorem to analyze an appropriately defined zero sum game, and then computing an approximate equilibrium strategy in the zero sum game.

Let H be the hypothesis class used by the weak learning algorithm A. We define a zero sum game as follows:

1. The action space for the minimization player (the "Data Player") is the set of datapoints in the dataset: $A_1 = D$.
2. The action space for the maximization player (the "Learner") is $A_2 = H$.
3. The cost function is C is defined as $C((x_i, y_i), h) = \mathbb{1}[h(x_i) = y_i]$.

How well can the players do in this game? The assumption that there exists a weak learning algorithm for D immediately lets us compute the min max value for the game — i.e. how well the learner could do if she got to best respond to a fixed strategy of the data player:

$$\begin{aligned}
\min\max(C) &= \min_{w\in\Delta[n]} \max_{h\in H} \sum_{i=1}^{n} w_i C((x_i, y_i), h) \\
&= \min_{w\in\Delta[n]} \max_{h\in H} w_i \mathbb{1}[h(x_i) = y_i] \\
&= \min_{w\in\Delta[n]} \max_{h\in H} acc(h, D, w) \\
&\geq 0.51
\end{aligned}$$

where the final inequality follows from the assumption that we have a weak learner that uses hypotheses in H.

We can now apply the minimax theorem to conclude that the Learner can do just as well, even if she is forced to commit to her strategy first:

$$\min\max(C) = \max\min(C) = \max_{p\in\Delta H} \min_{i\in[n]} \sum_{h\in H} p_h \mathbb{1}[h(x_i) = y_i] \geq 0.51$$

In other words, there is some *fixed* distribution p^* over hypotheses $h \in H$ such that for *every* data point $(x_i, y_i) \in D$, at least 51% of the probability mass under p is on hypotheses that correctly label (x_i, y_i). How can we use this? Consider the following "majority vote" classification rule f_{p^*}:

$$f_{p^*}(x) = \mathbb{1}\left[\sum_{h:h(x)=1} p^*_h \geq 0.5\right]$$

f_{p^*} turns out to have perfect accuracy.

Lemma 7 *For the distribution* $p^* = \max_{p \in \Delta H} \min_{i \in [n]} \sum_{h \in H} p_h \mathbb{1}[h(x_i) = y_i]$*, the hypothesis* f_{p^*} *satisfies* $acc(f_{p^*}, D) = 1$

Proof We need to show that for *every* $(x_i, y_i) \in D$, $f_{p^*}(x_i) = y_i$. From our minimax calculation, we know that for every i, $\sum_{h \in H} p_h \mathbb{1}[h(x_i) = y_i] \geq 0.51$. Hence if $y_i = 1$, we have that

$$\sum_{h \in H} p_h \mathbb{1}[h(x_i) = 1] = \sum_{h: h(x)=1} p^*_h \geq 0.51$$

and hence by definition $f_{p^*}(x_i) = 1$. Similarly, if $y_i = 0$, we know that $\sum_{h: h(x)=1} p^*_h < 0.49$ and hence by definition $f_{p^*}(x_i) = 0$, which completes the proof. ■ So we know there exists a hypothesis $(f_{p^*}(x_i))$

with zero training error — to complete the proof of our theorem, we only need to give an efficient algorithm for finding it. Note that in our proof of Lemma 7, the only property we used about p^* is that it was a distribution p which satisfied $\min_{i \in [n]} \sum_{h \in H} p_h \mathbb{1}[h(x_i) = y_i] > 0.5$. p^* satisfied this with some slack (0.01). But this means we would have the same result if we could compute $\hat{p}$, an ϵ-approximate max min strategy for our zero-sum game C will suffice, for $\epsilon < 0.01$. Fortunately, we derived an algorithm for computing approximate equilibria in zero sum games last class! Lets recall it here in our context. Remember that we need one player (here we will choose the minimization/data player) to maintain a distribution w^t over their actions using the polynomial weights update algorithm. The maximization player (the learner in our case) needs to be able to compute a cost maximizing action — an action that obtains value at least v in the game against the minimization players current strategy w^t, where v is the max min value of the game that we are aiming to achieve. For us, $v = 0.51$, and the corresponding computational task is exactly the weak learning problem. We assume we have access to a weak learner A, and so we can use it in our algorithm as a subroutine.

Algorithm 1 Boost(D, A)

```
Let $T \leftarrow \frac{4 \log n}{\epsilon^2}$ for $\epsilon < 0.01$.
Initialize a copy of polynomial weights to run over $w^t \in \Delta^n$.
for $t = 1$ to $T$ do
    Let $h^t = A(D, w^t)$
    Let $\ell^t \in [0,1]^m$ be such that $\ell^t_i = \mathbb{1}[h^t(x_i) = y_i]$.
    Pass $\ell^t$ to the PW algorithm.
end for
Let $\hat{p} = \frac{1}{T} \sum_{t=1}^{T} e_{h^t}$. (Note that this is concisely representable even though $H$ is large, because $\hat{p}$ has support over only the $T$ models $h^t$.)
Return $f_{\hat{p}}(x)$.
```

It remains to examine the running time of our Boosting algorithm. Since ϵ is a constant, on a dataset of size n, it runs for only $O(\log n)$ many iterations. At each iteration it makes a single call to our weak learning algorithm A, which we assume is polynomial time. It then has to update the polynomial weights distribution over the n datapoints, which takes time $O(n)$. Thus the total running time is $O(\log n(n + R(A))$, where $R(A)$ is the running time of our weak learning algorithm. Thus if our weak learning algorithm runs in polynomial time, so does our strong learning algorithm. ■ A couple of

remarks are in order:

1. First, as noted earlier, we have focused here on the problem of finding a hypothesis that correctly labels the training set, and have ignored the statistical issue of "generalization" — how well the learned hypothesis fits new data. This is a field of study in its own right, but by and large, for "simple" hypothesis classes H, fitting the training data implies coming close to fitting new data

drawn from the same distribution. Since H only needs to contain a weak learner, H can often taken to be something very simple, like depth 2 decision trees.

2. The hypothesis $f_{\hat{p}}$ that we output is not in H itself — but it is not too much more complex, in a way that can be formulated: namely, it is a threshold function that operates on a convex combination of at most $O(\log n)$ models from H. Hence, if H is a "simple" class, then $f_{\hat{p}}$ is also "simple" in a way that can be formalized to bound generalization error.

3. The assumption that there exists a weak learning algorithm is (as we have just shown!) much stronger than it first appears — in particular, it implies that the data can be perfectly classified. But recall that our analysis of our equilibrium computation algorithm last class made no assumptions on the action set of the maximization/best response player. Hence the analysis goes through even if for the $O(\log n)$ iterations of the algorithm, our learning algorithm A happens to be able to find models h^t that perform better than random guessing — even if it is not guaranteed to be able to do so for all possible distributions. Hence Boosting is a sensible – and popular — approach to learning in practice, even when the strong assumption of the existence of a weak learning algorithm does not technically hold.

CIS 320: Algorithms November 17, 2021

Lecture 21

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Calibrated Prediction

Suppose you turn on your local morning show, and the weatherman tells you that tomorrow there is a 10% chance of rain in your neighborhood. What does this mean? Tomorrow will only happen once, so this is not a repeatable event. If it rains, this is not an indictment of the weatherman — he did allow that there was some chance that it would. So how can you distinguish between a weatherman who knows what he is doing, from one who does not?

Lets write down a simple model — the weather prediction game. In rounds $t = 1$ to T:

1. The prediction player predicts some probability p_t of rain, for $p_t \in \{1/m, 2/m, \ldots, (m-1)/m, 1\}$.
2. The outcome $y_t \in \{0, 1\}$ is revealed: it either rains ($y_t = 1$) or it does not ($y_t = 0$).

Lets think about devising a test to determine whether the weatherman knows what he is doing. First, what should this mean? Suppose that there was really some true probabilistic process that governed rain, that the weatherman was privy to: every day, a probability p_t^* was revealed to the weatherman, and then it rained with that probability: $\Pr[y_t = 1] = p_t^*$. We would want that a weatherman who predicted $p_t = p_t^*$ every day should pass the test. Lets call this the oracular weatherman. It should also be possible to fail the test.

Here is a first attempt:

Definition 1 (Average Consistency) *A prediction strategy satisfies ϵ average consistency if for every sequence of outcomes, the sequence of predictions it generates $(p_1, y_1, \ldots, p_T, y_T)$ satisfies*

$$\mathrm{E}\left[\left|\frac{1}{T}\sum_{t=1}^{T} p_t - \sum_{t=1}^{T} y_T\right|\right] \leq \epsilon$$

We say it satisfies average consistency if $\epsilon \to 0$ as $T \to \infty$.

Certainly the oracular weatherman would pass this test, but its also clear that this is not stringent enough, because the following strategy ("The yesterday weatherman") also passes the test: "On day 1, predict $p_t = 0$, and on day t, predict $p_t = y_{t-1}$". i.e. just always predict that what happened yesterday will happen today. In this case we have $\left|\frac{1}{T}\sum_{t=1}^{T} p_t - \sum_{t=1}^{T} y_T\right| = y_T/T \leq 1/T$.

But it is easy to differentiate the yesterday weatherman from the oracular weatherman. If the oracular weatherman predicted a 100% chance of rain, it would *always* rain on such days. But the yesterday weatherman frequently predicts a 100% chance of rain and is wrong. In other words, the yesterday weatherman violates *prediction conditioned average consistency.* We'll bucket the weatherman's predictions into 100 buckets (i.e. by percentage points), and we'll say that a prediction p_t is in bucket i ($p_t \in B(i)$) if it is closer to $i/100$ than any other point $j/100$.

Definition 2 *Given a sequence of predictions and outcomes $(p_1, y_1, \ldots, p_T, y_T)$, let $n_T(i) = |\{t : p_t \in B(i)\}|$ be the number of rounds on which the prediction was in bucket i. The sequence satisfies ϵ-prediction conditioned average consistency for a bucket i if:*

$$\left|\frac{\sum_{t:p_t \in B(i)} y_t - p_t}{n_T(i)}\right| \leq \epsilon$$

In other words, conditioned on making a prediction of a $\approx i/100$ probability of rain, the weather forecaster should have been right — i.e. on the days on which he predicted a $\approx i/100$ probability of rain, it should have rained roughly a $i/100$ fraction of the time.

Thus suggests a stronger test: *calibration.* The idea is to ask for prediction conditioned average consistency for *all* 100 buckets i. But if we think about this a bit harder we realize that the oracular weatherman might not be able to satisfy this. Suppose there is only a single day for which $p_t^* \in B(30)$, and that as luck would have it, on that day it actually rained? A single stroke of bad luck (that happens 30% of the time!) would ruin conditional average consistency for $i = 30$. However, we can ask that this condition hold on average over the buckets i, weighted by their frequency:

Definition 3 *A prediction strategy satisfies ϵ-average calibration if for all sequences of outcomes, the sequence of predictions it generates $(p_1, y_1, \ldots, p_T, y_T)$ satisfies:*

$$\mathrm{E}\left[\sum_{i=1}^{100} \frac{n_T(i)}{T} \cdot \left|\frac{\sum_{t:p_t \in B(i)} y_t - p_t}{n_T(i)}\right|\right] = \frac{1}{T}\mathrm{E}\left[\sum_{i=1}^{100}\left|\sum_{t=1}^{T} \mathbb{1}[p_t \in B(i)](y_t - p_t)\right|\right] \leq \epsilon$$

We say it satisfies average calibration if $\epsilon \to 0$ as $T \to \infty$

It will be more convenient to instead work with a "Euclidean" metric of calibration error:

$$L_T = \sum_{i=1}^{100}\left(\sum_{t=1}^{T} \mathbb{1}[p_t \in B(i)](y_t - p_t)\right)^2$$

You can confirm (this is the "Cauchy-Schwartz inequality") that the average calibration loss ϵ of a strategy is upper bounded by:

$$\epsilon \leq \mathrm{E}\left[\frac{10}{T}\sqrt{L_T}\right] \leq \frac{10}{T}\sqrt{\mathrm{E}[L_T]}$$

It turns out there is an algorithm that will let any weatherman pass the calibration test as well, even without any knowledge of weather. To design the algorithm, lets suppose our weatherman has already made predictions up through day $s-1$, and is considering what he should predict on day s. If he predicts $p_s \in B(i)$ and the outcome turns out to be y_s, then the increase in the loss function will be:

$$\begin{aligned}
\Delta_s(p_s, y_s) &= L_s - L_{s-1} \\
&= \left(\sum_{t=1}^{s} \mathbb{1}[p_t \in B(i)](y_t - p_t)\right)^2 - \left(\sum_{t=1}^{s-1} \mathbb{1}[p_t \in B(i)](y_t - p_t)\right)^2 \\
&= \left(V_{s-1}^i + (y_s - p_s)\right)^2 - \left(V_{s-1}^i\right)^2 \\
&\leq 2V_{s-1}^i \cdot (y_s - p_s) + 1
\end{aligned}$$

where $V_{s-1}^i = \sum_{t=1}^{s-1} \mathbb{1}[p_t \in B(i)](y_t - p_t)$ is a fixed constant at the time that the weatherman must make her decision on day s. Observe that $|V_{s-1}^i| \leq T$.

Now suppose we could show that the weatherman had a distribution over predictions that would guarantee that $\mathrm{E}[\Delta_s(p_s, y_s)] \leq 2T/m + 1$ at every round. Then we would have that:

$$\mathrm{E}[L_T] = \sum_{t=1}^{T} \mathrm{E}[\Delta_t(p_t, y_t)] \leq \frac{2T^2}{m} + T = O\left(\frac{T^2}{m} + T\right)$$

and our calibration loss would be bounded by $\epsilon \leq \frac{10}{T}\sqrt{\mathrm{E}[L_T]} = O(\frac{1}{\sqrt{m}} + \frac{1}{\sqrt{T}})$. Hence if we chose $m = T$, we would have calibration loss on the order of $O(1/\sqrt{T})$, and therefore a predictions strategy satisfying average calibration.

So that's the plan. To understand how our algorithm should make predictions at round s Define a zero-sum game that has cost function taking value:

$$C_s(p, y) = 2V_{s-1}^i \cdot (y_s - p_s) + 1$$

for each $p \in B(i)$. The minimization player (The Learner) has action set $A_1 = \{1/m, 2/m, \ldots, 1\}$, and the maximization player (The Adversary) has action set $A_2 = \{0, 1\}$. We have by construction that $\Delta_s(p_s, y_s) \leq C_s(p_s, y_s)$, so we need to bound the value of this game. This is easier to do if the adversary moves first, because its much easier to make a prediction about something you already know! This corresponds the max min value of the game, in which the adversary first commits to a distribution $q \in \Delta\{0, 1\}$.

Note that once the adversary commits to a distribution q, this fixes $\mathrm{E}_{y\sim q}[y]$, which the Learner knows. So in this ordering of moves, the Learner is actually in the role of the oracular weatherman! To best respond, the Learner should set $p = \mathrm{E}[y]$, which would guarantee that $\mathrm{E}[C_s(p, y)] = 1$. The learner cannot necessarily quite do this (because his action set only contains multiples of $1/m$), but he can always find a p such that $|p - \mathrm{E}_q[y]| \leq 1/m$. Hence we have:

$$\max_{q\in\Delta A_2} \min_{p\in A_1} \mathrm{E}_{y\sim q}[C_s(p, y)] \leq 2 \cdot \frac{|V^i_{s-1}|}{m} + 1 \leq 2 \cdot \frac{T}{m} + 1$$

Applying the minimax theorem, we can conclude that the value of the game remains the same if the Learner moves first:

$$\min_{\hat{p}\in\Delta A_1} \max_{y\in A_2} \mathrm{E}_{p\sim\hat{p}}[C_s(p, y)] \leq 2 \cdot \frac{T}{m} + 1$$

In other words, at every round s, the learner has a distribution over predictions $\hat{p}_s$ that guarantees that *no matter what the label y_s is*:

$$\mathrm{E}_{p_s\sim\hat{p}_s}[\Delta_s(p_s, y_s)] \leq \max_{y\in A_2} \mathrm{E}_{p\sim\hat{p}}[C_s(p, y)] \leq 2 \cdot \frac{T}{m} + 1$$

Which is exactly what we have wanted. In other words, we have proven the following theorem:

Theorem 4 *There exists a prediction strategy that against an arbitrary adversarially chosen sequence of T outcomes satisfies ϵ-average calibration for $\epsilon = O(1/\sqrt{T})$*

What is that strategy? It simply plays the minmax equilibrium strategy for the Learner in the zero-sum game we derived above! We can always efficiently compute the equilibrium of a zero-sum game by writing it as a linear program which explicitely finds the distribution over actions for the learner that minimizes the maximum cost resulting from any action of the adversary:

Algorithm 1 Algorithm for Predicting at Round s

```
**Let** $\hat{p} \in \Delta[m]$ be the solution to the following linear program defined over variables $\hat{p}_1, \ldots, \hat{p}_T$:
```

Minimize γ such that:

$$\sum_{t=1}^{T} \hat{p}_t = 1, \quad \sum_{t=1}^{T} \hat{p}_t C_s\left(\frac{t}{T}, 0\right) \leq \gamma, \quad \sum_{t=1}^{T} \hat{p}_t C_s\left(\frac{t}{T}, 1\right) \leq \gamma$$

```
**Select** $p_s = \frac{t}{T}$ with probability $\hat{p}_t$.
```

There are only 3 constraints, and T variables in this linear program, so solving it takes time polynomial in T. In fact, in this particular case, the minmax equilibrium strategy for the learner has a nice closed form (that you may work out on the homework) that can be sampled from in time independent of T, with no need to solve a linear program.

A couple of remarks are in order:

1. Here the minimax theorem gave us an existential proof of the existence of an algorithm! We only needed to reason about the (easy) problem of predicting something about a distribution we already

know, because the adversary (who in this thought experiment is forced to announce his strategy) told us. The minimax theorem tells us we can do just as well in the actual case, in which we must commit to an algorithm first, without knowledge of the adversary's plans.

2. This argument was rather generic to any linear (i.e. based on bounding sums or averages) test aimed at distinguishing the oracular weatherman from a fraud. This is because the minimax theorem literally is allowing us to analyze the Learner as if she is the oracular weatherman!
3. We are able to mimic the oracular weatherman even if the truth is that outcomes are chosen adversarially, without any probabilistic model at all. This should make you think critically about how much we can learn from empirical tests of probabilistic models.