COMS E6998-9: Algorithms for Massive Data (Spring'19) April 18, 2019

Lecture Lecture 24: Learning augmented algorithms

Instructor: *Alex Andoni* Scribes: *Kiran Vodrahalli, Lei You*

1 Administrative and Scheduling

I hope you're all working on your projects. There are two components remaining: The final writeup and the presentation. There are 17 teams covering a range of topics touching most of the main subjects that we did in the class. I'm thinking of scheduling 10 minutes per presentation (each team has a slot of 10 minutes). You can present any way you want. On a team, one person is allowed to present everything. It's also okay to switch between presenters. The grade is per team. Classes are 75 minutes, so we can do 7 teams per lecture. This will be relatively tight since we will have only 5 minutes extra. So don't run late! Think of it as having 9 minutes to talk with maybe one question. 3 teams will be doing it next Thursday. If any of you volunteer, I'll be highly appreciative! The presentation doesn't have to go and say "these are the results" yet. They don't have to be at the level of the final writeup. You should present the problem and what has happened. The presentation is mostly judged on clarity and flow, you don't have to go into too much detail. It's good to cover some technical detail. I'll try to give you an example next Tuesday. Doing it a little early isn't such a big deal; you don't have to do everything.

There were some questions on Piazza regarding the projects. Reading projects should have some kind of message – some comparison, or historical timeline of a problem and related directions. Take max-flow for example: For a long time there were results that were not linear, but in the past decade there were $(1 + \epsilon)$ approximations with near-linear time. So a project would focus on roughly two papers, saying what existed before, and what the two key ideas were. Some papers are very deep and have lots of ideas, so just one or two is enough. If the paper is lighter on content, then look at more papers.

It is important to separate what is the main crucial idea that made this paper, versus many bells and whistles. Papers will get something working (very basic case that showcases the main idea) and then does a lot of generalization (this case, and this case, and so on). From the perspective of a survey, focus on the main idea and not the bells and whistles. It's more important to focus on one problem and go deep rather than add all generalizations.

Recall project proposal is 5%, presentation is 10%, and the writeup is 30%. Regarding the final writeup, I've uploaded two examples online (both on the research-y side). I'll try to upload (previous classes) survey type as well.

2 Learning augmented algorithms/data-driven algorithms

This is not hundred percent aligned with the class, but I think it fits.

Most of the class, what we are talking about is algorithms, with a classic approach from theoretical computer science, kind of a worst-case point of view. It's a very pessimistic view of the world, and it ignores that you might have some side information about the dataset. This is what we have done so far.

There are some existing approaches for moving away from worst-case.

2.1 Average-case Analysis

The oldest approach is called "average-case" analysis, and sometimes it is mentioned in analysis of algorithms. Your data is not an adversary, but it comes from nature, and nature is usually not adversarial. Maybe it comes from some distribution (some kind of average case). Example: consider sorting, where it's a random permutation. Sometimes this helps some algorithms, sometimes it doesn't. It's easier to deal with but average case requires you to assume some distribution over the data, which might be not true.

So we do not know: the distribution of the input, and small variations might screw up my algorithm. Another part is that sometimes, the input comes from other processes. These can give instances which are bad (e.g. not all data is generated by nature, maybe it's from some computer process, and could be adapting to the case where your algorithm is the slowest).

2.2 Instance-optimal analysis

We talked about this during distribution testing. In general we showed we could do it with $\mathcal{O}(\sqrt{n})$, where n is support size, number of samples, but for some distributions it's much easier to test this. For instance, Gaussian – there are algorithms which do this much faster. So we can say "for this particular distribution that I'm testing", I want to design an algorithm for testing this particular distribution. So the algorithm is somewhat data-adaptive.

2.3 Learning-agumented algorithms

I don't necessarily want to assume something about the data-set, it's not necessarily a bad dataset, but I don't really know much about the data. So how about as I take in the data, I look at it, and I adapt my algorithm to this data.

The model is roughly the following: We have some dataset $\mathcal{D}$ that I don't know much about, but from previous interaction, I've learned a little bit about this dataset. I have some kind of oracle or helper that gives me side information about this dataset. Let's call it a genie. This genie is some kind of procedure (ML model) that looks at the dataset, sees what kind of queries come, and tries to learn a bit about the dataset and gives some prediction of the answers. Let's be more specific here and talk about binary search.

Definition 1. *Binary search.*
An algorithm to find element x in a sorted array A.

Claim 2. *Worst-case for binary search.*
The worst case is $\mathcal{O}(\log n)$ time where $n = |A|$.

If my array is random numbers, there's very little in doing better. But say there's some structure. Say A contains an arithmetic progressiion $A_i = \alpha + i\beta$. So if you can learn these parameters α, β, then you can directly index into the array. Then queries take only constant $\mathcal{O}(1)$ time!

Maybe it's not an arithmetic progression. This is assuming very strong structure. But maybe there are datasets where things are not quite dataset progresrsion, but maybe there is close structure.

So maybe instead I'll do some kind of learning in advance, and I'll get a genie G which will give a me a prediction of where my query is. So what is the natural algorithm here?

Now take your theory hat off and you're at work. Your co-worker says they throw a deep neural network which lets you predict the index well. It turns out there are implementations of B-trees which use this kind of principle, so it's not unrealistic.

So what do you? How do you improve your binary search algorithm? Just pick $G(x)$, and if it fails, run normal binary search. We can't completely trust the genie – it gives a good prediction, but it can be a little off. Maybe it's completely off because of adversarial examples! So G isn't good enough by itself, it's not robust enough maybe. We want some guarantee that the algorithm still works. If the genie is very good then it should improve our time, but if it's bad, it shouldn't damage us too much.

Definition 3. *Learning-augmented binary search.*

1. *Get $i = G(x)$.*
2. *Check if $A_i = x$.*
3. *WLOG, $A_i < x$ (if $A_i > x$, do it in the other direction). Check A_{i+2^j} vs. x, $j = 1, 2, \cdots$ until $A_{i+2^j} > x$.*
4. *Binary earch on $A[i, \cdots, i + 2^j]$.*

Now the question is what do we think about the improvement in runtime? If the genie is good, we should do it much faster, if the genie is bad, then in the worst case we should do $\log n$ runtime. We'd like to quantify this.

We can assign some kind of quality to the genie. Then we can use this quality in the runtime. This is a kind of assumption for the analysis.

Suppose G has some η error: For any $x \in A$, $|G(x) - i| < \eta$ s.t. $A_i = x$.

Then,

Claim 4. *The algorithm runs in time $2\lceil\log_2 \eta\rceil$.*

Proof. Can be seen from the fact that error η reduces the length of the array we search to a block of size η. □

So we didn't completely strictly improve – the standard binary search runs in time $\log_2 n$. In the worst case η is n, so maybe we lost factor 2 in the runtime. So if the genie is bad, we probably shouldn't be doing this.

In general we want a few properties of a genie:

1. Independence: The algorithm should be independent of the model of the genie. We don't want to hard-code in the algorithm the notion that a genie has a particular structure – if the genie changes a little bit, you might break the algorithm. It also means you can change the genie easily.
2. Consistency: Better genie, better algorithm.
3. Robustness: Our legacy from worst-case analysis: The algorithm should not be much worse than worst-case.

3 Online algorithm: ski-rental

Online algorithms have a different model of efficiency. Here is how it goes. It's called ski-rental for the following reason:

You go skiing – you have two options, you rent skis for a dollar a day, or you decide to buy the skis and don't pay anything for rental. If you ski a few days, then you should just rent, but if you ski enough, you should probably buy. But you don't know at the beginning – maybe you'll decide you hate skiing.

So this is the hard part of the problem – how many days in advance will you be skiing? You only find this out on the day you decide not to ski.

If this problem was invented today, maybe it would be called "cloud rental" – startup, should I buy a bunch of cluster computing, or rent from Amazon cloud for a while, maybe I'll sell company before I decide to buy a cluster. This also ignores the issues that money now is better than money later and so on.

Another example is apartment rental – should I continue renting my apartment or should I buy one?

Let's say x is the day you decide to stop skiing. This is unknown until day x. Let b be the cost to buy, and let 1 be the cost to rent per day.

Total cost at the end, if you decide to buy: $D + b$, where D is number of days rented.

The performance of algorithm that we measure is just how much we paid. We want to characterize how well we did – we'll just compare to the best possible strategy, to someone who knew in advance how many days they would be skiing. So, OPT = best strategy that knows x in advance. This is just $\min(x, b)$.

We measure this with competitive ratio:

Definition 5. *Competitive ratio.*
Let I be a problem instance (e.g., fixing an x in ski-rental). $C_A(I) = \frac{\text{cost achieved by algorithm Aon I}}{\text{OPT cost}}$

The goal is to get an algorithm A which minimizes over A $\max_I C_A(I)$. This is a classic algorithm and it has been analyzed.

Definition 6. *D. Algorithm .*
Strategy: Just rent until day $b-1$, at day b buy.

What is $C_A(x)$ for this algorithm? If $x < b$, the cost is just x. Otherwise, we will pay $b - 1 + b$ since we rented for $b-1$ days before buying on day b. If $x < b$, we have ratio 1. If $x \geq b$, then cost is $\frac{2b-1}{b} \leq 2$. So this D-algorithm has competitive ratio 2 since 2 is an upper bound on $C_A(I)$.

Claim 7. *This is optimal for a deterministic algorithm.*

Proof. A deterministic algorithm must choose a day to stop renting and buying. This is what any deterministic algorithm looks like.

Suppose this algorithm buys on day y. WLOG, $y \leq b$. (If it buys on day b, then the competitive ratio is at least 2). So then, the adversary (bad distribution) will set $x = y$. So you go until the day when you buy, and when you buy, you stop skiing. So in this case the competitive ratio will be $\frac{y+b}{y} \geq 2$. □

Claim 8. *You can do better if you have randomized algorithms (don't decide on a specific day, but decide on a random day where randomness is independent of x). You can get competitive ratio which is about $\frac{e}{e-1} \approx 1.58$.*

Proof. Our algorithm is just to decide to buy on day $y \sim p_i = (1-1/b)^{b-1} \cdot \frac{1}{b(1-(1-1/b)^b)}$, where $i \in [b]$. So it picks this day y, and rents until day y (or stops). Otherwise it buys on day y. □

The worst case is if you stop skiing on the day you buy. So if there's randomness, it's hard to build a specific case where you stop skiing on eactly the day.

It's possible to deduce this probability distribution by writing out the expected competitive ratio and choosing the one that minimizes the expected competitive ratio, and this is the best one.

3.1 Learning-augmented version

So this is the classic work thus far, and for the worst case situation. Maybe I have some kind of estimate how far I can go or how many days to ski? In particular there is a genie that tells me a guesstimate of x. We will call y the guesstimate of our genie. The genie could be good or bad, we want to parametrize our algorithm as a function of how good this genie guesstimate is. Define error $\eta = |x-y|$. Perhaps the competitive ratio can now depend on η as well.

We had that we want consistency and robustness. Let's define these properties quantitatively.

Definition 9. *We say that our algorithm is α-consistent if our competitive ratio is at most α when $\eta = 0$.*

Definition 10. *We say our algorithm is β-robust if competitive ratio is $\leq \beta$ for any η. Ideally it will go down with η.*

There will be a little bit of a tension between these two things.

Definition 11. *Algorithm I.*
Rent until the day y (genie's prediction) if $y < b$, or if $y \geq b$, buy on day 1.

Claim 12. *Cost of algorithm I is $\leq OPT + \eta$.*

Proof. If $y \geq b$ and $x \geq b$ or $x, y < b$, then our decision coincides with the good decision. The algorithm value is then equal to OPT.

If $y \geq b$ and $x < b$: This is a little bit bad. The optimal strategy is to rent until day x, whereas the genie predicts you'll ski more than in reality, so you buy. The value of the algorithm is $b \leq y \leq x + \eta = OPT + \eta$. So you overpay the optimal algorithm by a quantity of η.

Similarly $y < b$ and $x \geq b$, then the algorithm value will be equal to renting until day x, which is $x \leq b + (x-b) \leq OPT + \eta$.

Do we have consistency? If $\eta = 0$, then competitive ratio is 1 (we achieve OPT) – so it's $\alpha = 1$ consistent.

What aobut robustness and the genie is really bad. What's the worst possible competitive ratio? It's $\max_\eta \frac{OPT+\eta}{OPT} = \infty$, which is NOT robust!

□

So what was the issue? We completely trusted the genie and threw out any algorithmic strategy. We didn't have any algorithmic strategy. So the natural idea is do whatever we did before the algorithm. So we'll make a choice when to stop renting and buy, and this choice will be informed by the genie.

Let's introduce a parameter $\lambda \in (0,1)$ which is a measure of trust in the genie. We set $y = G$. Buy on day $\lceil \lambda b \rceil$ if $y \geq b$, and buy on day $\lceil b/\lambda \rceil$ if $y < b$.

What does this say? We buy a bit earlier than if we were to completely ignore the genie in the first case. Wait a bit longer in the latter case since genie says we won't be skiing for too long.

Claim 13. *The cost is at most* $\min(1+\lambda+\frac{\eta}{(1-\lambda)OPT}, \frac{1-\lambda}{\lambda})$.

Proof. Similar case analysis. $\square$

Let's do our consistency and robustness analysis. We have $\alpha = 1+\lambda$ consistency. What is β? The worst possible η makes the terms go to ∞, and we're left with $\beta = \frac{1+\lambda}{\lambda}$. So we have a tradeoff between dataset being nice and bad.

If $\lambda \to 0$, then it's consistent, but not robust. If $\lambda \to 1$, then both quantities are roughly 2, which is like the first deterministic algorithm.

This is the genie-augmented version of the deterministic algorithm.

There should also be a genie-augmented version of the randomized algorithm.

Definition 14. *Genie-augmented randomized algorithm.*
If $y \geq b$, *then let* $k = \lfloor \lambda b \rfloor$. *Buy on day* $\sim q_i = (1-1/b)^{k-1}\frac{1}{b(1-(1-1/b)^k)}$. *Otherwise,* $\ell = \lfloor b/\lambda \rfloor$. *Then buy on day* $\sim w_i = (1-1/b)^{\ell-1}\frac{1}{b(1-(1-1/b)^\ell)}$.

Then we have this algorithm is $\frac{\lambda}{1-e^{-\lambda}}$-consistent and $\frac{1}{1-e^{-(\lambda-1/b)}}$-robust for $\lambda > 1/b$.

Think of λ as a constant and b as tending to infinity.