Machine Learning for Algorithms Design (TTIC 31290) **Fall 2025**

Lecture 15: Algorithms with Predictions (November 20, 2025)

Instructors: Avrim Blum and Dravyansh Sharma Lecturer: Vaidehi Srinivas

1 Introduction

The goal of *algorithms with predictions* (also called *learning-augmented algorithms*) is to use extra **unreliable predictions** to improve the performance of algorithms.

1.1 Motivating examples

Keep in mind the following two examples.

1. [LV21] **Competitive Caching with Machine Learned Advice**

   We are maintaining a cache of size k, and handling a sequence of page requests. When a page is requested, if it is in the cache we can serve it for free. If it is not in the cache, we *evict* some page in the cache, load the requested page into the cache at cost 1. The goal is to design an eviction strategy that minimizes the cost, a.k.a. the number of "cache misses."

   Classically we can design a randomized algorithm that can achieve a competitive ratio of $O(\log k)$ for this problem. However, in practice, we may expect that the sequence of page requests may be predictable (for example, if we know something about the process that is running). [LV21] show how to incorporate imperfect information about future page requests to make the algorithm more efficient.

2. [DMVW23] **Predictive Flows for Faster Ford-Fulkerson**

   We are solving an instance of max-flow via Ford-Fulkerson. Recall that this algorithm proceeds iteratively. On each iteration, it finds a path to push one more unit of flow from the source to the sink. This allows us bound the number of iterations by the amount of flow in the solution.

   However, we may have some information about what the flow solution looks like. For example, if each day we must solve a routing problem on a road network, and the road network changes very little from day to day, we may have a good sense of what most of the solution looks like before we begin. [DMVW23] show how to use a guess for what the max-flow solution will be, to run Ford-Fulkerson in a fewer number of iterations. In particular, if the guess is good, it could speed up the algorithm significantly. If the guess is bad, then it will be no worse than running Ford-Fulkerson from scratch.

1.2 Comparison to Data-Driven Algorithms

This setting is very related to data-driven algorithms, which you have seen before in this course. Here are some typical similarities and differences of these areas.

Data-Driven Algorithms	Algorithms With Predictions
• Algorithm given a few "parameters"	• Algorithm given high-dimensional "prediction," often from a black box source
• **Goal:** learn best parameter setting for a distribution of instances	• **Goal:** show that good predictions increase performance significantly

These two perspectives on algorithm design with machine learning are complementary. Whether you call the extra information "parameters" or "predictions" is a matter of perspective. In an ideal case, you would want both of these types of guarantees simultaneously: that you can learn information, and that the information you learn will improve the performance of your algorithm significantly.

This will be addressed in your next lecture on "learning the predictions for algorithms with predictions."

1.3 Framework for Algorithms with Predictions

An algorithms with predictions guarantee is typically consists of three desiderata.

(1) **Consistency**: when predictions are good, achieve the best possible performance that takes advantage of the extra information

(2) **Robustness**: never perform worse than the best algorithm that ignores the prediction entirely, even when the prediction is misleading

(3) **Graceful degradation**: performance of the algorithm trades-off gracefully in the magnitude of the prediction error

Remark. *While these three goals are typically consistent across the algorithms with predictions literature, the terminology is a source of spirited disagreement ☺. You may also see the same three goals referred to as* consistency*,* competitiveness*, and* robustness*.*

The third notion of **graceful degradation** is important. Consider the earlier example of solving max-flow with Ford-Fulkerson. It is easy to achieve consistency and robustness. First, check if the predicted solution is perfect. This is fast, since it is much faster to verify that there are no augmenting flows left, than to solve an instance from scratch. If the prediction is not perfect, solve the instance from scratch. This is unsatisfying. Even if the prediction is almost perfect, we get no benefit from this approach.

A strength of the above framework is its versatility in being applicable to many different areas of algorithm design. Indeed there are lines of work developing algorithms with predictions in

mechanism design, *online algorithms*, *streaming algorithms*, *fast algorithms*, *dynamic algorithms*, *differential privacy*, and more!

This means that there are many important modeling decisions required to instantiate this framework to a new problem. Here are some important questions to consider and address when approaching a new problem.

- **Performance**: what does performance mean for your problem? (e.g. time complexity, space complexity, decision-making cost, etc.)
- **Prediction**: what information are we asking to be predicted? Is it something we could reasonably predict in practice? Can it improve the performance of an algorithm significantly?
- **Prediction error metric**: how do you measure error in an imperfect prediction? This is highly tied to **graceful degradation**. Is it reasonable to get low error predictions in practice? Is this learnable in theory? Is it learnable in practice?

2 Review: Classical Caching

In this lecture we present the learning-augmented caching strategy from [LV21]. In this section, we introduce the caching problem and review the classic *marking algorithm*, which achieves the best possible competitive ratio.

2.1 Problem

The *caching problem* (also called *paging problem*) is a classical problem in online algorithms.

We have a cache of size k. There is a universe of m items. On each day, an element e is requested:

- If e is in the cache, the request is served with cost 0.
- If e is not in the cache, the algorithm must bring e into the cache, by *evicting* an item currently in the cache, and writing e in its place. This has cost 1.

The goal of the algorithm is to serve a sequence of requests while minimizing the total cost.

Example. *Cache starts in configuration* $\boxed{a}\boxed{b}\boxed{c}\boxed{d}\boxed{e}$.

1. *Request:* c. *It is in the cache, cost to algorithm* $= 0$.
2. *Request:* f. *Not in the cache. Algorithm chooses to evict* a*, cost to the algorithm* $= 1$. *New state of cache is* $\boxed{f}\boxed{b}\boxed{c}\boxed{d}\boxed{e}$.
3. *Request:* b. *It is in the cache, cost to algorithm* $= 0$.
4. *Request:* a. *Not in the cache. Algorithm chooses to evict* b*, cost to the algorithm* $= 1$. *New state of cache is* $\boxed{f}\boxed{a}\boxed{c}\boxed{d}\boxed{e}$.

We analyze the performance of an algorithm via *competitive analysis*. That is, we bound the approximation ratio, compared to the *best strategy in hindsight* (what the best strategy would have done if it knew the full sequence of requests in advance).

In the above example, if we knew the request sequence would be c, f, b, a, we could have achieved only one cache miss on f. Thus the competitive ratio α is bounded by

$$\alpha \geq \frac{\text{cost}(\mathsf{ALG})}{\text{cost}(\mathsf{OPT})} = \frac{2}{1} = 2.$$

We will analyze competitive ratio against an *oblivious adversary*. That is, we assume that the adversary has committed to an (unknown) sequence of requests before our algorithm starts making potentially randomized decisions. Thus any randomness that our algorithm uses is independent of future requests that the adversary will make.

2.2 The Marking Algorithm

The marking algorithm achieves the best possible competitive ratio for this problem.

Marking algorithm [FKL+91]:
For new request e:

- If e not in cache:
  - If all items in the cache are marked, unmark all items in cache
  - Evict a uniformly random unmarked element from the cache
- Mark e

2.3 Analysis

Theorem 2.1 (From [FKL+91])**.** *The marking algorithm is $O(\log k)$ competitive against an oblivious adversary.*

Remark. *It is actually known how to analyze the competitive ratio of this algorithm up to constant factors (without any $O(\cdot)$ analysis). We will do a less precise analysis, since our goal is to look at the algorithms with predictions version of this.*

Proof. Define a *phase* of the marking algorithm as the time between two subsequent unmarkings.

To give a competitive ratio, we need to upper bound the number of mistakes that our algorithm makes, and lower bound the number of mistakes that OPT, the best in hindsight, must make. We begin by analyzing OPT.

On phase i, define

- ℓ_i, the number of "clean" items requested in phase i– items that are requested in that phase, that were not yet requested in this phase or the previous phase.

- $d_{\text{init}} = |\mathsf{OPT}\text{ cache } \setminus \mathsf{OPT}\text{ cache }|$ at the start of phase i
- $d_{\text{fin}} = |\mathsf{OPT}\text{ cache } \setminus \mathsf{OPT}\text{ cache }|$ at the end of phase i.

We know that the marking algorithm will incur a cache miss for every clean item. We want to argue that OPT also has to make mistakes on many of the clean items.

First, we observe that

$$\mathsf{OPT}\text{ cache misses on phase } i \geq \ell_i - d_{\text{init}},$$

because ALG had to miss all ℓ_i of the clean items, and OPT can miss at most d_{init} of them.

We can also observe that

$$\mathsf{OPT}\text{ cache misses on phase } i \geq d_{\text{fin}}.$$

This is because at the end of phase i, ALG has exactly the elements that were requested in phase i in the cache. Thus, for every different element that OPT has in cache, it must have evicted at least one of the elements that was requested in this phase from the cache.

Now, we can lower bound the number of mistakes by the max of these two, but it is easier to work with the average.

$$\mathsf{OPT}\text{ cache misses on phase } i \;\geq\; \max\{\ell_i - d_{\text{init}}, d_{\text{fin}}\} \;\geq\; \frac{\ell_i - d_{\text{init}} + d_{\text{fin}}}{2}.$$

Summing over all phases, we get

$$\begin{aligned}
\mathsf{OPT}\text{ total cache misses } &\geq \frac{1}{2} \sum_{\text{phase } i} \left(\ell_i - d_{\text{init}}^{(i)} + d_{\text{fin}}^{(i)}\right) \\
&\geq \frac{1}{2} \left(-d_{\text{init}}^{\star} + d_{\text{fin}}^{\star} + \sum_{\text{phase } i} \ell_i\right) \\
&\geq \frac{1}{2} \sum_{\text{phase } i} \ell_i,
\end{aligned}$$

where in the last step we assume that the cache of ALG and OPT start out empty, so $d_{\text{init}}^{\star}$ at the beginning of the entire sequence is 0.

Now that we have established that OPT incurs cache misses on essentially half of the clean items, we will charge the number of cache misses that ALG makes to the number of clean items.

On phase i, ALG sees:

- ℓ_i clean items
- $k - \ell_i$ other items, which we will call "stale" items. These items were seen in the previous phase, and are unmarked in the cache at the beginning of the phase.

There is a traditional analysis that one can do here that allows us to derive the expected number of mistakes that the algorithm makes up to constant factors. We will instead provide a coarser analysis, that will generalize better to the algorithms with predictions setting.

Within a phase i, we will think of ALG's evictions as happening in "eviction chains." When a clean item e_1 is requested, it begins an eviction chain. If it evicts an item e_2 that will not be requested in this phase, the eviction chain ends. Otherwise, when e_2 is requested, it needs to come back into the cache, so it will evict some other item, and the eviction chain continues.

We know that there are ℓ_i different eviction chains started by the different clean items. So if we can bound the expected length of any eviction chain, we can bound the expected number of evictions in the phase.

At the beginning of phase i, imagine that the cache is laid out in order of when items will be requested next, with earlier requests coming earlier in the order.

e_1	e_2	e_3	e_4	e_5	$\dots$	e_k

Now, we analyze what happens to one evictions chain. When a clean item c arrives to start the eviction chain, some subset of these original unmarked items will still be unmarked in the cache. Others have been replaced and/or marked.

e_1	e_2	~~e_3~~ $\to m_1$ ✓	e_4	~~e_5~~ $\to m_2$ ✓	$\dots$	e_k

Of these, the algorithm will choose one uniformly at random to replace with c. Suppose it chooses e_2.

e_1	~~e_2~~ $\to c$ ✓	~~e_3~~ $\to m_1$ ✓	e_4	~~e_5~~ $\to m_2$ ✓	$\dots$	e_k

Now, if the eviction chain continues, it is because e_2 has been requested. Because we have laid out the cache in order of when requests happen, this means that when e_2 is requested, all spots before e_2 have already been requested, so they have been marked. In addition, it is possible that some other spots after e_2 have also been marked.

e_1 ✓	~~e_2~~ $\to c$ ✓	~~e_3~~ $\to m_1$ ✓	e_4	~~e_5~~ $\to m_2$ ✓	$\dots$	e_k

When e_2 chooses some element to evict, it will choose uniformly at random from the elements that are still unmarked, that must come after it in the cache.

e_1 ✓	~~e_2~~ $\to c$ ✓	~~e_3~~ $\to m_1$ ✓	~~e_4~~ $\to e_2$ ✓	~~e_5~~ $\to m_2$ ✓	$\dots$	e_k

Thus, for every new element in the eviction chain, the number of unmarked elements remaining in the cache will go down by at least a factor $\frac{1}{2}$ in expectation. Since the cache is of size k, by a quicksort analysis, this means that the expected number of items in the chain is $O(\log k)$.

Since there are ℓ_i clean items that start eviction chains, the expected number of evictions that ALG makes in phase i is $\ell_i \cdot O(\log k)$. Summing over all phases, we can bound the competitive ratio of marking by

$$\frac{\text{cost}(\text{ALG})}{\text{cost}(\text{OPT})} \leq \frac{O(\log k) \sum_{\text{phase } i} \ell_i}{\sum_{\text{phase } i} \ell_i} \leq O(\log k).$$

■

3 Caching With Predictions

On one hand, $O(\log k)$ is a good competitive ratio, since it is the best that we can achieve in the worst case. On the other hand, we cannot actually tolerate this many cache misses in practice.

This motivates an algorithms with predictions approach. If inputs that we see in practice do not look adversarial, then perhaps we can learn information that helps us do better.

3.1 What prediction should we ask for?

We want to choose a prediction that is both something we could expect to predict (up to some error), and that will help our algorithm make better decisions.

If we had all of the information in the world, that is, if we had access to the whole sequence of requests up front, then what would we do?

Theorem 3.1 (Bélády's Algorithm)**.** *The algorithm that always evicts the item in cache that will be requested again farthest in the future is offline optimal.*

Proof sketch. Consider any other sequence of evictions. We can always swap the eviction of an item that will return sooner with the evicting an item that will return later, without increasing the number of cache misses of the sequence. ■

This suggests a natural prediction to ask for. When an item arrives, we ask for a prediction of when it will be requested next.

This is as far as we got in class. The rest of the notes are for completeness.

3.2 A First Attempt

Now that we are armed with the prediction, the first thing we might try is to run the Bélády rule (evict latest returning element) with the predicted order in which the elements will return.

Unfortunately, with this strategy, even small errors in the prediction can cause large costs to the algorithm.

Exercise 3.2. *Give a true sequence and predicted sequence that causes the unmodified Bélády's rule to incur high cost. Even better, show such a true sequence and predicted sequence that are not very different (prediction is "low error"), which causes a high cost to the unmodified Bélády rule.*

3.3 Refined Approach

What we would like is to incorporate the benefit of the predictions through some use of the Bélády rule, while also ensuring the worst-case competitive ratio of the Marking algorithm. It would be helpful if we could detect when the predictions were incorrect, and switch between the Bélády rule and the marking algorithm. This motivates [LV21]'s approach.

Marking With Predictions [LV21]:
For new request e:

- If e not in cache:
  - If all items in the cache are marked, unmark all items in cache
  - If e is in an eviction chain of fewer than $\log k$ evictions, then evict the unmarked item in the cache that is predicted to reappear latest in the future. Else, evict a uniformly random unmarked element from the cache.
- Mark e

Now, we want to show two properties.

(1) **Consistency + Graceful Degradation:** If the predictions are low error, then marking with predictions will perform well.

(2) **Robustness:** No matter what, the algorithm will never perform worse than Marking without predictions.

Lemma 3.3 (Consistency + Graceful Degradation of Marking with Predictions)**.** *Marking with Predictions with incur number of cache misses at most*

$$2\ \mathrm{cost}(\mathsf{OPT}) + O(1) \cdot \mathrm{error}$$

where error *is the number of inversions between the predicted arrival sequence and the true arrival sequence.*

Remark. *Here we consider the error metric on the predictions to be the number of inversions between the true arrival sequence and the predicted arrival sequence. That is, it is the number of pairs of requests e_1, e_2, such that e_1 arrived before e_2, but e_2 was predicted to arrive before e_1. We choose this to give the most accessible proof, and we note that [LV21] handles more general error metrics.*

Proof. We prove the contrapositive: if Marking with Predictions incurs high cost, then it must be that the predictions had high error.

The number of cache misses incurred by Marking with Predictions in some phase i is equal to the total length of the eviction chains of cache misses on phase i.

Consider one of these eviction chains. We look at the first part of the eviction chain in which the evictions were done according to the Bélády rule. Let e_2 be an element in the cache, that was evicted by some element e_1, according to the Bélády rule with the predictions. Then in the future e_2 is requested, must rejoin the cache, and evicts an element e_3.

When e_2 is requested, e_3 is unmarked, so e_2 is requested sooner than e_3 in the true sequence. However, when e_1 was requested, the Bélády rule evicted e_2 instead of e_3. Thus e_2 was predicted to be requested later than e_3. Thus, e_2 and e_3 were predicted to arrive in the wrong order.

This tells us that the first $\log k$ items in the chain which were evicted according to the Bélády rule must have been predicted to arrive in the reverse order from which they actually did. Thus the

true arrival order and the predicted arrival order differ by at least as many inversions as the length of the chain (minus the clean item that started the chain).

Now consider the latter part of the chain that had evictions done according to the random rule. If the random rule is invoked on this chain, it means that the chain had already reached length $\log k$ using the Bélády rule. By the same argument as the classical proof, after the random rule kicks in, the chain will grow by $O(\log k)$ in expectation.

Thus, the total length of the chain is at most $O(1)$ times the number of inversions in the predicted arrival times, among the items in the chain in this phase.

The total number of cache misses in phase i is then

$$\ell_i + O(1) \cdot (\#\text{ inversions of predictions for requests during phase } i)\,.$$

This allows us to bound the total number of cache misses over all phases as

$$\leq 2\ \text{cost}(\mathsf{OPT}) + O(1) \cdot \text{error},$$

where error is the number of inversions between the predicted arrival sequence and the true arrival sequence. ■

Lemma 3.4 (Robustness of Marking with Predictions)**.** *Marking with Predictions is always $O(\log k)$ competitive, regardless of the prediction quality.*

Proof. Like in the analysis of the standard marking algorithm, we can bound the number of cache misses by the total length of the eviction chains over all of the phases.

The first Bélády rule phase can only increase the length of any chain by at most $\log k$. After the chain switches to the random eviction rule, the same analysis still holds to show that the chain will grow by $O(\log k)$ in expectation.

Thus the length of any eviction chain is still $O(\log k)$, and the competitive analysis of the standard Marking algorithm goes through. ■

Finally, we can put together the two consistency + graceful degradation guarantee and robustness guarantee to get our final theorem.

Theorem 3.5 (Marking with Predictions [LV21])**.** *Marking with Predictions incurs number of cache misses upper bounded by*

$$\min\left\{2\ \text{cost}(\mathsf{OPT}) + O(1) \cdot \text{error},\ O(\log k) \cdot \text{cost}(\mathsf{OPT})\right\},$$

where where error *is the number of inversions between the predicted arrival sequence and the true arrival sequence.*

References

[DMVW23] Sami Davies, Benjamin Moseley, Sergei Vassilvitskii, and Yuyan Wang. Predictive flows for faster ford-fulkerson. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[FKL+91] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, December 1991.

[LV21] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4), July 2021.