Various Models of Online Algorithms with Predictions

Sanjay Gollapudi, Lance Mathias, Chethan Bhateja

May 2023

Abstract Existing works [12, 13] have explored the use of predictions to improve the performance of online algorithms, especially with the advent of machine learning which allows us to construct effective oracles in practice. In this survey, we examine several frameworks which build upon the basic framework of prediction-augmented online algorithms: parsimonious predictions, randomly infused advice, and multiple experts. These frameworks introduce additional considerations, enabling algorithms to be more practical in the real world.

1 Introduction

Online algorithms have been extensively studied for many years and the field is continuously growing. We have extensively covered Online algorithms and various methods of analysis of their competitive ratios. However, in all of the examples in class, the information coming online was completely random and we had to make decisions, which were possibly irrational. What if the information was not completely random? What if there was some prediction that gives us some clue as to what the information coming online is? Would this allow for better competitive ratios when compared to online algorithms with no predictions? Further, given an instance of a problem, what should we predict? Some problems have many sources of randomness which can be predicted, so correctly predicting a significant source of randomness is crucial when trying to achieve competitive ratios. In general, if a prediction is error-free, our online algorithm should be as efficient as OPT, which is usually defined as the offline optimal algorithm, and if our prediction is arbitrarily bad, our online algorithm (with predictions) should not be much worse than a regular online algorithm.

While online algorithms with predictions are relatively new with the rise of machine learning, there are new models still being explored. One could imagine that generating predictions for each step of the online algorithm could be costly and slow. What if we were to limit the number of predictions generated in each step; instead of using n predictions per step, we use $k < n$ predictions? Does using fewer predictions provide quantitatively similar competitive ratios? In addition to this model, there are a few other models of online algorithms with predictions we will explore in this survey.

Before we go over some simple examples of online algorithms with predictions there are a couple of terms we need to introduce.

Definition 1. *We say an algorithm A is C-competitive if for all inputs σ:*

$$cost(A(\sigma)) \leq C \cdot cost(OPT(\sigma)) + O(1).$$

We say C is the competitive ratio.

As we stated earlier we need our online algorithm to perform as well as OPT when our prediction is error-free. In other words, our online algorithm needs to be consistent.

Definition 2 (Consistency)**.** *We say an algorithm A, is α-consistent if the competitive ratio tends to α as the error, η tends to 0.*

We also stated earlier that our online algorithm needs to perform as well as the online algorithm without predictions when the prediction has an arbitrarily large error. In other words, our online algorithm also needs to be robust.

Definition 3 (Robustness)**.** *We say an algorithm A, is β-robust if the competitive ratio is bounded by β as the error is arbitrarily bad.*

Definition 4. *A metric space is an ordered pair (M, d) where $d : M \times M \to \mathbb{R}_{\geq 0}$ which satisfies the following properties:*

- $d(x, x) = 0$.
- *If $x \neq y$, then $d(x, y) > 0$.*
- $d(x, y) = d(y, x)$.
- $d(x, z) \leq d(x, y) + d(y, z)$, *otherwise known as the triangle inequality.*

2 Basic Examples

The field of augmenting online algorithms with machine-learned predictions has been around for years [13]. In this section, we highlight two newer algorithms that have not been described in previous surveys [12].

2.1 Minimum Spanning Tree

The minimum spanning tree is a well-studied problem with various efficient offline algorithms like Kruskal's and Prim's. We consider the online version of the minimum spanning tree. However, we have to be careful with the information which comes online. If the vertices or edges of a graph $G = (V, E)$ ($|V| = n, |E| = m$) are coming online and we must make an irreversible decision of whether or not to include the vertex/edge, we are forced to add every vertex/edge. This is because any vertex/edge could be the only way to access a component of the graph (think of a vertex with degree 1). As a result, we consider the problem where we know the graph G up front but the weight of the edges comes online. We can define the error of our prediction as follows. Let $p_i := |\hat{w}(e) - w(e)|$ where $\hat{w}(e)$ is the predicted weight of edge e and $w(e)$ is the actual weight of e. Then sort the errors p_i such that $p_{i_1} \geq p_{i_2} \geq \cdots \geq p_{i_m}$. Then the error

$$\eta(\hat{w}, w) = \sum_{j=1}^{n-1} p_{i_j}.$$

Since our spanning tree has $n - 1$ edges, the error is the sum of the $n - 1$ largest individual errors of the predicted edge weights. It turns out the simplest algorithm achieves the best consistency: follow the prediction or FTP. Berg et. al. [1] showed that FTP is 1-consistent.

2.2 k-server

The k-server problem is defined as follows: requests come online at certain points in our metric space and we are tasked with moving one of our k servers to fulfill this request. We want to minimize the total distance traveled by all the k-servers after handling all of the requests. Further, we limit our metric space to the metric space on the line. To be more specific, let $(s_1, s_2, \cdots s_k) \in \mathbb{R}^k$ denote not only the names of our k servers but also their positions on the line. Let $C_t = (s_1, s_2, \cdots s_k)$ denote the configuration of our k servers after the t^{th} request. If $\sigma = (r_1, r_2, \cdots, r_n)$ is our request sequence, then the cost of fulfilling r_t is $d(C_{t-1}, C_t)$ which is the distance the servers traveled to get from C_{t-1} to C_t (C_0 is our initial configuration of our servers). Therefore, our objective function can be written as

$$\min \sum_{t=1}^{n} d(C_{t-1}, C_t).$$

To create our prediction model, for each request r_t we would have a prediction $p_t \in \{1, \cdots, k\}$ indicating which server should fulfill the request. For example, $p_t = 1$ would mean s_1 should fulfill our request. Call the algorithm which follows the predictions as true FTP. Then for a request sequence σ the error $\eta = FTP(\sigma) - OPT(\sigma)$

There is a very well-known algorithm called the double coverage algorithm which has been shown to have the best competitive ratio k. For a given request r_t such that $s_i \leq r_t \leq s_{i+1}$ for some i, the algorithm will move both s_i and s_{i+1} a distance of $\min\{d(s_i, r_t), d(s_{i+1}, r_t)\}$ towards r_t.

We can change the algorithm to adapt to our access to predictions. The new algorithm, which is called LambdaDC [9], works as follows: for a given $\lambda \in [0, 1]$ assume $s_i \leq r_t \leq s_{i+1}$ for some i. Then if $p_t <= i$, we move s_i with speed 1 and s_{i+1} with speed λ, both towards r_t, until either s_i, s_{i+1} hits r_t. Similarly, if $p_t \geq i + 1$ then we move s_i with speed λ and s_{i+1} with speed 1 until either s_i, s_{i+1} hits r_t. In comparison, the double coverage algorithm always moved $s_i$and s_{i+1} with speed 1 towards r_t. If $r_t \leq s_1$ or $r_t \geq s_k$ just moves s_1 or s_k towards r_t respectively.

While the analysis can be done for k-servers, we will look at the 2-server case. Through a potential function argument, the following can be shown.

Theorem 1. *For any parameter $\lambda \in [0, 1]$* LambdaDC *has a competitive ratio of at most*

$$\min\{1 + \frac{1}{\lambda}, (1 + \lambda)(1 + \frac{\eta}{OPT})\}.$$

This would imply, for 2 servers, LambdaDC *is $1 + \lambda$-consistent and $1 + \frac{1}{\lambda}$-robust.*

3 More Refined Prediction-Augmented Settings

While the model of prediction-augmented algorithms is powerful, it ignores some strategies and challenges present when applying online algorithms in the real world. These include cost and limited availability of predictions and the ability to use multiple predictors, leading to a few more specific problem settings we present below.

3.1 Parsimonious Predictions

Depending on what is predicted and how a problem is structured, at each algorithm step, an online algorithm may query predictions for multiple objects, such as cache pages, jobs, or servers. Predictions can be expensive and we may wish to predict on only a few objects, leading to the setting of **parsimonious predictions**, where we aim to use as few predictions as possible at each algorithm step.

3.1.1 Application to Caching

This setting was first described by Im et al. [8], who applied it to the problem of online caching with predictions. McGeoch and Sleator [11] proved that without predictions, it is impossible to achieve a competitive ratio better than H_k where H_k is the k-th harmonic number.

Lykouris & Vassilvitskii [10] considered a k-page cache in the prediction-augmented setting and proposed an algorithm that is 2-consistent and $2 + \log k$-robust, but queries the predicted next request time for all k pages at each timestep. Im et al. realized they could get by with fewer predictions and proposed an algorithm that queries predicted next request time for only $b < k$ pages, achieving competitive ratio $O(\log_{b+1} k)$-consistency while maintaining $O(\log k)$-robustness [8].

Recall marking algorithms for caching, in which pages are marked when they are used, only unmarked pages are evicted, and all pages are unmarked when there are no pages left to mark. Each minimal request sequence from all unmarked pages to all marked pages is called a **phase**, a page not requested in the previous phase is called **clean**, and a page that was requested is called **stale**.

As a warm-up, Im et al. consider an error-free oracle that perfectly predicts the next request time. As in the offline caching setting, consider each phase in the sequence of requests. Let U be the set of unmarked pages and τ_p be the predicted eviction time for a page p. Im et al. propose Algorithm 1:

Algorithm 1 Naive Eviction with Oracle

```
if |U| ≥ εk then
    return p ∼ uniformly from U
else
    if We have not queried in this phase then
        for p ∈ U do
            Query τ_p
        end for
    end if
    return p_* ← argmax_{p∈U} τ_p
end if
```

They give the following proof on the performance and number of queries of this algorithm [8].

Theorem 2. *For any $\epsilon > 0$ and request sequence Γ, this naive eviction algorithm is $O(\log(1/\epsilon))$-competitive and makes at most $\epsilon|\Gamma|$ queries.*

Proof. Consider a single phase of the above marking algorithm and let c be the number of requests for clean pages in this phase, so that there are $k - c$ requests for stale pages. Now let $p_1, \ldots, p_k$ be the pages in the cache at the start of this phase, sorted in order of when they are requested in this phase and breaking ties arbitrarily. In particular, this means that if $i < j \leq k - c$, then p_i is requested before p_j, while if $i > k - c$, p_i is not requested in this phase.

First, consider the first request to a page p_i where $i \leq k - \epsilon k$ and let $c^{(i)}$ be the number of clean requests in this phase p_i is requested. We will calculate the probability that this request incurs a cache miss. Before this request, there have been $i-1$ requests to the stale pages $p_1, \ldots, p_{i-1}$ which are now in the cache, leaving $k - i + 1$ stale pages that could potentially be missing due to evictions. Since there have been $c^{(i)}$ clean requests so far, $c^{(i)}$ of these pages are actually missing and as evictions are uniformly random among unmarked pages, the probability p_i incurs a cache miss is $\frac{c^{(i)}}{k-i+1} \leq \frac{c}{k-i+1}$.

Next, consider p_i where $i > k - \epsilon k$. Note that after the request to $p_{k-\epsilon k}$, the algorithm queries the next request times for all ϵk unmarked pages and evicts the page requested furthest in the future, which must be one of the pages $p_{k-c}, \ldots, p_k$ that are not requested in this phase. Then p_i can only incur a cache miss if it was evicted before $p_{k-\epsilon k}$ was requested. At most c of these ϵk pages can be evicted so p_i incurs a cache miss with probability at most $\frac{c}{\epsilon k}$.

Then by linearity of expectation, the expected total number of cache misses due to stale pages in this phase is at most

$$\sum_{i=1}^{k-\epsilon k} \frac{c}{k-i+1} + \sum_{i=k-\epsilon k}^{k-c} \frac{c}{\epsilon k} \leq c(H_k - H_{\epsilon k}) + c$$

. There are c misses from clean pages so the total number of misses is $O(c \log \frac{1}{\epsilon})$. Fiat et al.[6] show that the cost of OPT is linear in the number of clean requests, so the algorithm obtains the desired competitive ratio of $O(\log \frac{1}{\epsilon})$.

Since there are at most $\frac{|\Gamma|}{k}$ phases with at most ϵk queries per phase, the total number of queries is at most $\epsilon|\Gamma|$. □

In fact, the adaptive query algorithm that Im et al. propose later in the paper is even simpler to describe than the naive algorithm: query b unmarked pages uniformly randomly and evict the page with the next request time predicted furthest in the future. Here b can be used to adapt between reducing queries and achieving a good competitive ratio. Though the analysis of this algorithm is more tedious, Im et al. bound

Algorithm 2 Adaptive Query Eviction

$S \leftarrow b$ pages from the unmarked pages U sampled U.A.R. without replacement
Query predictions τ_p for all $p \in S$
return $p_* \leftarrow \text{argmax}_{p \in S} \tau_p$

the number of misses by analyzing eviction chains, chains of pages that successively evict each other in each phase. Bounding the expected length of these chains of misses in terms of b, k, and prediction error, they find this algorithm achieves competitive ratio $O(\log_{b+1} k + \mathbb{E}[\eta])$, where η is the l_1-error of their predictions. Through a simple modification of switching back to uniformly random evictions when eviction chains become long, they are able to also obtain a worst-case bound of $O(\log k)$, the best possible competitive ratio to the offline optimal algorithm.

3.1.2 Application to Online Linear Optimization

Next, we consider the online convex optimization problem against an oblivious adversary introduced in [3]. We are given some convex domain D which the problem is defined over. For each timestep $t \in \{1, \ldots, T\}$, our algorithm makes a prediction $x_t \in D$ and the adversary reveals a cost vector c_t. Our algorithm then incurs a loss of $\langle x_t, c_t \rangle$. We define the regret of our algorithm with respect to some reference u as

$$R(c, u) = \sum_{t=1}^{T} \langle c_t, x_t \rangle - \langle c_t, u \rangle$$

In other words, we compare our algorithm to a reference that plays some fixed move u for all timesteps t. We define the regret of the algorithm as

$$R(c) = \sup_u R(c, u)$$

Note that since the inner product $\langle x_t, c_t \rangle$ can be negative, it's possible to achieve negative regret. For simplicity, we restrict our examples to optimization over a d-dimensional ℓ_2 ball. Furthermore, Bhaskara et. al. define a quantitative way to describe whether a prediction is good:

Definition 5 (α-goodness)**.** *For some timestep t, a prediction h_t is α-good if $\langle h_t, c_t \rangle \geq \alpha \|c_t\|$. A prediction that is not α-good is said to be bad.*

Algorithms such as Online Gradient Descent yield $O(\sqrt{T})$ regret without using any predictions. Bhaskara et. al. [2] devised an algorithm that achieves $O(\frac{1}{\alpha} \log T)$ regret when given one prediction at each timestep, assuming all predictions are good. If up to B predictions are bad, the algorithm can still achieve $O\left(\frac{1}{\alpha} \log T + \sqrt{\frac{B}{\alpha} \log T}\right)$ regret. In a subsequent paper, Bhaskara et. al. [3] describes an improved algorithm that achieves $O(\frac{1}{\alpha} \log T)$ regret if all predictions are good, or $O(\frac{\sqrt{B}}{\alpha} \log T)$ if B predictions are bad. In all cases, this algorithm uses only $O(\sqrt{T})$ predictions. Furthermore, they showed that the bound on the number of predictions required is sharp; any algorithm that uses $o(\sqrt{T})$ predictions will have $\Omega(\sqrt{T})$ regret, even if every prediction is arbitrarily good.

3.1.3 Application to the Multi-Armed Bandit Problem

The Multi-Armed bandit problem is defined as follows: we have n arms, with each arm having its own probability distribution of success. At each step, we can choose one of the n arms which will give an unknown reward. The goal is to maximize your payout/reward over all the steps. Without access to predictions/hints, we would have to essentially guess and check the different arms to gain information on which arms give the best rewards. However, they do not use competitive ratios to compare the optimal solution. Rather they use regret.

To formally describe the problem, we are given a set A of n arms and T rounds, which are both known, and for each arm a there is a reward distribution D_a. In each round $t \in [T]$, we pick an arm a_t. Then a reward $r_t \in [0, 1]$ is sampled independently from the distribution D_{a_t} and is collected. Further, since our rewards are selected randomly, let $\mu_a := \mathbb{E}[D_A]$. Then we can define $\mu^* := \max_{a \in A} \mu_a$. Then notice that the OPT will always select the arm which gives the highest expected reward. In other words, $OPT = T\mu^*$. Then we can define the regret R as follows:

$$R = T\mu^* - \sum_{t=1}^{T} \mu_{a_t}.$$

Note that in each round, we make a decision based on the results of the previous rounds. If we had access to predictions for the rewards the arms would give, we would be able to be more efficient with our guesses and gain more rewards if the predictions are correct. But once again, generating n predictions for each arm for every round would be very costly. Bhaskara et. al. [4] introduced various policies where we have the ability to probe $k \leq n$ of the arms to predict their respective rewards. In particular, they analyzed the following 2 policies:

- BESTPROBE: At each time step t, this policy will query a set S_t of at most $k \leq n$ arms and return the arm which will give the best reward (the actual reward value is not disclosed). In other words, if the reward of arm a at time step t is r_{a_t}, then this policy will return $R = \max_{a_t \in S_t} r_{a_t}$. Note that in this policy, while you are making k probes, you only make use of the best probe.
- ALLPROBE: At each time step t, this policy will not only give you the arm which will give the best reward but will also give you the actual reward value for every $a \in S_t$. This policy will also return $R = \max_{a_t \in S_t} r_{a_t}$.

Bhaskara et. al. showed an upper bound on the regret of $O(n^2 \log n)$ when $k = 2$ for the BESTPROBE policy. Further, some of the predictions utilized in each step could be incorrect or faulty. If B of the T probes utilized in the BESTPROBE model are faulty, then they showed an upper bound on the regret of $O(n^2 \log n \sqrt{B+1})$. Lastly, they showed an upper bound on the regret of $O(n^2)$ when $k = 3$ for the ALLPROBE policy.

3.2 Randomly Infused Advice (RAI)

Emek et. al. [5] propose the framework of *randomly infused advice*, in which rather than explicitly soliciting a predictor for advice, machine-learned predictions are incorporated directly into the random bit stream used by a randomized algorithm. In particular, each time the randomized algorithm queries random bits, the bits are replaced with a (correct) machine-learned prediction with probability α, called the *infusion parameter*. Additionally, it's assumed that the algorithm cannot store past predictions.

This framework has two main advantages over other frameworks:

1. Many randomized online algorithms, such as caching and online set cover, do not require any modification to incorporate infused predictions.
2. The RAI framework can easily account for unreliable predictions, e.g. predictions that are wrong with some (known) probability, simply by adjusting the infusion parameter.

3.2.1 Application to Caching

Again consider the online paging problem with an oblivious adversary, and suppose we have a cache of size k. Emek et. al. showed in [5] that the randomized marking algorithm, with no modification other than randomly infused advice, achieves a competitive ratio of $\min\{2H_k, \frac{2}{\alpha}\}$ where H_k is the k-th harmonic number, or approximately $\log k$, and α is the infusion parameter. In this case, the infused prediction gives a recommendation on which specific element to evict.

3.3 Multiple Predictors

Another important aspect of predictions in the real world is that we often have multiple predictors available, similar to scenarios such as the classical experts' problem. Some recent works have focused on how to leverage this information from multiple predictors in settings with prediction-augmented online algorithms.

3.3.1 Application to Ski Rental

The multiple predictors setting was first described by Gollapudi et al. [7], who apply it to the ski rental problem. As a reminder, in the ski rental problem, we are skiing for some unknown number of days x and can rent skis for \$1 per day or buy them for \$b. Our goal is to minimize cost.
Gollapudi et al. adopt the usual robustness definition but define consistency as follows:

Definition 6 (α-consistent)**.** *In the k predictor setting, let the best predictor have error Δ. Then an algorithm is α-**consistent** if for all inputs it incurs cost $ALG < \alpha(OPT + \Delta)$.*

This consistency definition only requires the best predictor to be perfect, allowing others to be erroneous. This highlights a challenge of this multiple predictor setting, which is to zero in on good predictors.
Gollapudi et al. [7] first consider the consistency setting where there is at least one perfect predictor. Let $k = 2$ and let the predicted numbers of days be a_1, a_2. When $a_1, a_2 < b$ or $a_1, a_2 \geq b$, we know one of these predictions is correct and can simply buy or rent respectively t achieve optimality. However, if the predictions disagree more severely, for example, if $a_1 < b, a_2 \geq b$, simple strategies such as always renting or buying lead to an unbounded competitive ratio. They suppose for simplicity that $a_2 >> b$ and note that two strategies make sense: buying immediately or renting up today a_1 and buying after. Given a number of days x, the worst case competitive ratio for these two strategies is $\frac{b}{x}$ (bad when x is "small") and $\frac{x+b}{x}$ (bad when $x > b$) is respectively. To obtain the lowest possible competitive ratio, Gollapudi et al. solve the equation $\frac{b}{x} = \frac{x+b}{b}$, finding where these two functions cross. The solution is $x = (\phi - 1)b$ so we should rent if $a_1 < x$ and buy otherwise. They come up with an example to show this algorithm has a competitive ratio ϕ. Extending to k predictors, they find a similar system of equations key breakpoints $x_1, \ldots, x_{k-1}$

$$\frac{b}{x_1} = \frac{b + x_1}{x_2} = \frac{b + x_2}{x_3} = \cdots = \frac{b + x_{k-2}}{x_{k-1}} = \frac{b + x_{k-1}}{b}$$

They propose an algorithm to continue renting and buy only if we encounter an interval $(x_i, x_{i+1}$ without any predictions. Here, they find that the algorithm achieves a competitive ratio of γ_k, the k-acci constant

instead of the Fibonacci (2-acci) constant.
To bound the competitive ratio in terms of the best prediction error Δ, Gollapudi et al. make a slight modification of buying in the middle of empty breakpoint intervals instead of at the beginning. Through casework on the possible numbers of days, they bound the competitive ratio of this new algorithm by $O(\gamma_k(\text{OPT} + \Delta))$.
Regarding robustness, Gollapudi et al. show the following:

Lemma 3. *If the earliest time an algorithm for this problem can buy given any prediction is λb, it has robustness factor $\beta = 1 + \frac{1}{\lambda}$.*

Proof. If an algorithm buys at λb, in the worst case this was the last day, incurring competitive ratio $1 + \frac{1}{\lambda}$. Then the worst competitive ratio, which determines robustness, is determined by the earliest possible buying time λb. □

Given this, Gollapudi et al. simply modify the algorithm to not buy during the first λb days, where λ can be chosen as desired to tune between robustness and consistency. Experimentally, they find the addition of the first few predictors significantly improve performance and that robustness becomes vital as predictor error increases.

3.3.2 Multi-shop Ski Rental (MSSR)

In later work, Wang et al. [14] tackle the problem of multi-shop ski rental. Suppose instead of one store, we now have n stores to choose from. We denote the price to rent from store i as r_i, and the price to buy skis from store i as b_i. At the beginning of the trip, we choose a single store to rent or buy from; on subsequent days, we must continue to rent or buy from that same store. Wang et. al. also assume that the rental and buying prices follow the pattern $0 < r_1 < \cdots < r_n$, and $b_1 > \cdots > b_n > 0$.

The authors first introduce a deterministic algorithm to solve MSSR, described as follows: We first choose the store $i^* = \arg\min(r_i + (b_i - r_i)/b_n)$. Then, the skier rents for $b_n - 1$ days and buys on day b_n. The authors show that this algorithm yields a competitive ratio of $r_i + (b_i - r_i)/b_n$, and prove that this is optimal if no predictions are used.

The authors then introduce a randomized algorithm that uses a single prediction, y, on how long the ski trip will be. Notably, if $y \geq b_n$, we choose shop n, otherwise we choose shop 1. We then sample a number j from a probability distribution parameterized by the costs of the chosen shop, and rent for the first $j - 1$ days, buying on day j. The authors prove that this algorithm is $\frac{r_n\lambda}{1-\exp(-r_n\lambda)}$-consistent and $\frac{b_1}{b_n}\max\{\frac{r_n}{1-\exp(-r_n(\lambda-1/b_n))}, \frac{1/\lambda-1/b_1}{1-\exp(-1/\lambda)}\}$-robust, where $\lambda \in (1/b_n, 1)$ is a hyperparameter we choose denoting how much we distrust our predictions.

Finally, the authors propose a randomized algorithm that has access to m predictions $y_1, \ldots, y_m$ at each step. The authors define a sum of indicators

$$z = \sum_{i=1}^{m} \mathbf{1}\{y_i \geq b_n\}$$

We choose shop n if $z \geq \frac{m}{2}$, otherwise we choose shop 1. As before, we sample a number j from a probability distribution parameterized by the costs of the chosen shop, and rent for the first $j - 1$ days, buying on day j. The authors prove that this algorithm is $\frac{r_n\lambda}{1-\exp(-r_n(\lambda/(m+1)-1/b_n))}$-consistent and $\frac{b_1}{b_n}\max\{\frac{r_n}{1-\exp(-r_n(\lambda/(m+1)-1/b_n))}, \frac{m+1/\lambda-1/b_1}{1-\exp(-1/\lambda)}\}$-robust, where $\lambda \in (1/b_n, 1)$ is defined as before.

The authors perform empirical model validation to confirm that increasing the number of predictions m improves the competitive ratio when predictors have small error, and hurts the competitive ratio when predictors have large error.

References

[1] M. Berg, J. Boyar, L. M. Favrholdt, and K. S. Larsen. Online minimum spanning trees with weight predictions, 2023.

[2] A. Bhaskara, A. Cutkosky, R. Kumar, and M. Purohit. Online linear optimization with many hints, 2020.

[3] A. Bhaskara, A. Cutkosky, R. Kumar, and M. Purohit. Logarithmic regret from sublinear hints, 2021.

[4] A. Bhaskara, S. Gollapudi, S. Im, K. Kollias, and K. Munagala. Online learning and bandits with queried hints, 2022.

[5] Y. Emek, Y. Gil, M. Pacut, and S. Schmid. Online algorithms with randomly infused advice, 2023.

[6] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *CoRR*, cs.DS/0205038, 2002.

[7] S. Gollapudi and D. Panigrahi. Online algorithms for rent-or-buy with expert advice. In K. Chaudhuri and R. Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2319–2327. PMLR, 09–15 Jun 2019.

[8] S. Im, R. Kumar, A. Petety, and M. Purohit. Parsimonious learning-augmented caching, 2022.

[9] A. Lindermayr, N. Megow, and B. Simon. Double coverage with machine-learned advice, 2021.

[10] T. Lykouris and S. Vassilvitskii. Competitive caching with machine learned advice, 2020.

[11] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, June 1991.

[12] M. Mitzenmacher and S. Vassilvitskii. Algorithms with predictions, 2020.

[13] M. Purohit, Z. Svitkina, and R. Kumar. Improving online algorithms via ml predictions. In *Neural Information Processing Systems*, 2018.

[14] S. Wang, J. Li, and S. Wang. Online algorithms for multi-shop ski rental with machine learned advice, 2020.