Competitive Caching with Machine Learned Advice

Thodoris Lykouris [1] **Sergei Vassilvitskii** [2]

Abstract

We develop a framework for augmenting online algorithms with a machine learned oracle to achieve competitive ratios that provably improve upon unconditional worst case lower bounds when the oracle has low error. Our approach treats the oracle as a complete black box, and is not dependent on its inner workings, or the exact distribution of its errors. We apply this framework to the traditional caching problem creating an eviction strategy for a cache of size k. We demonstrate that naively following the oracle's recommendations may lead to very poor performance, even when the average error is quite low. Instead we show how to modify the Marker algorithm to take into account the oracle's predictions, and prove that this combined approach achieves a competitive ratio that both (i) decreases as the oracle's error decreases, and (ii) is always capped by $O(\log k)$, which can be achieved without any oracle input. We complement our results with an empirical evaluation of our algorithm on real world datasets, and show that it performs well empirically even using simple off-the-shelf predictions.

1. Introduction

Despite the success and prevalence of machine learned systems across many application domains, there are still a lot of hurdles that one needs to overcome to deploy an ML system in practice (Sculley et al., 2015). As these systems are rarely 100% perfect, a key challenge is dealing with errors that inevitably arise.

There are many reasons that learned systems may exhibit errors when deployed. First, most of them are trained to be good *on average*, minimizing some expected loss. In doing so, the system may invest its efforts on reducing the error on the majority of inputs, at the expense of increased error on a handful of outliers. Another problem is that generalization error guarantees only apply when the train and test examples are drawn from the same distribution. If this assumption is violated, either due to distribution drift or adversarial examples (Szegedy et al., 2014), the machine learned predictions may be very far from the truth. In all cases, any system backed by machine learning needs to be robust enough to handle occasional errors.

While machine learning is in the business of predicting the unknown, online algorithms provide guidance on how to act without *any* knowledge of the future. These powerful methods show how to hedge decisions so that, regardless of what the future holds, the online algorithm performs nearly as well as the optimal offline algorithm. However these guarantees come at a cost: since they protect against the worst case, online algorithms may be overly cautious, which leads to high competitive ratios even for seemingly simple problems.

In this work, we ask: *What if the online algorithm is equipped with a machine learned oracle? How can one use this oracle to combine the predictive power of machine learning with the robustness of online algorithms?*

We focus on a prototypical example of this area: the online paging, or *caching* problem. In this setting, requests arrive one at a time to a server equipped with a small amount of memory. Upon processing a request, the server places the answer in the memory (in case an identical request comes in the future). Since the local memory has limited size, the server must decide which of the current elements to evict. It is well known that if the local memory or *cache* has size k, then any deterministic algorithm has competitive ratio $\Omega(k)$. However, an $O(k)$ bound can be also achieved by almost any reasonable strategy, thus this metric fails to distinguish between algorithms that perform well in practice and those that perform poorly. The competitive ratio of the best randomized algorithm is $\Theta(\log k)$ which, despite far from trivial, is also much higher than observed in practice.

In contrast, we show how to use machine learned predictions to achieve a competitive ratio of approximately $2 + O(\min(\sqrt{\eta/\textsc{Opt}}, \log k)$, when using a hypothesis with total absolute loss η (see Section 3 for a precise statement of

*Equal contribution [1]Cornell University, Ithaca, NY, USA [2]Google Research, New York, NY, USA. Correspondence to: Thodoris Lykouris <teddlyk@cs.cornell.edu>, Sergei Vassilvitskii <sergeiv@google.com>.

Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018. Copyright 2018 by the author(s).

the results). Thus, when the predictions are accurate (small η), our approach circumvents the worst case lower bounds. On the other hand, even when the oracle is inaccurate (large η), the performance degrades gracefully to almost match the worst case bound.

1.1. Our Contributions

The conceptual contribution of the paper lies in formalizing the interplay between machine learning and competitive analysis by introducing a general framework (Section 2), and a set of desiderata for online algorithms that use a machine learned oracle.

We look for approaches that:

- Make *minimal* assumptions on the machine learned oracle. Specifically, since most machine learning guarantees are on the expected performance, our results are parametric as a function of the error in the oracle η, and not the distribution of the error.
- Are *robust*: a better oracle (one with lower η) should result in lower competitive ratios.
- Are worst-case *competitive*: no matter the performance of the oracle on the particular instance, the algorithm should behave comparably to the best online algorithm for the problem.

We instantiate the general framework to the online caching problem, specifying the prediction made by the oracle, and presenting an algorithm that uses that prediction effectively (Section 3). Along the way we show that algorithmic innovation is necessary: simply following the recommendations of the predictor may lead to poor performance, even when the average error is small (Section 3.1). Instead, we adapt the Marker algorithm (Fiat et al., 1991) to carefully incorporate the feedback of the predictor. The resulting approach, which we call *Predictive Marker* has guarantees that capture the best of both worlds: the algorithm performs better as the error of the oracle decreases, but performs nearly as well as the best online algorithm in the worst case.

Our analysis generalizes to multiple potential loss functions (such as absolute loss and squared loss). This freedom in the loss function with the black-box access to the oracle allows our results to be strengthened with future progress in machine learning and reduces the task of designing better algorithms to the task of finding better predictors.

We complement our theoretical findings with empirical results (Section 4). We test the performance of our algorithm on public data using off-the-shelf machine learning models. We compare the performance to the Least Recently Used (LRU) algorithm, which serves as the gold standard, the original Marker algorithm, as well as directly using the predictor. In all cases, the Predictive Marker algorithm outperforms known approaches.

Before moving to the main technical content, we provide an example that highlights the main concepts of this work.

1.2. Example: Faster Binary Search

Consider the classical binary search problem. Given a sorted array A on n elements and a query element q, the goal is to either find the index of q in the array, or state that it is not in the set. The textbook method is binary search: compare the value of q to that of the middle element of A, and recurse on the correct half of the array. After $O(\log n)$ probes, the method either finds q or returns.

Instead of using binary search, one can train a classifier, h, to predict the position of q in the array. (Although this may appear to be overly complex, Kraska et al. (2017) empirically demonstrate the advantages of such a method.) How to use such a classifier? A simple approach is to first probe the location at $h(q)$; if q is not found there, we immediately know whether it is smaller or larger. Suppose q is larger than the element in $A[h(q)]$ and the array is sorted in increasing order. We probe elements at $h(q)+2, h(q)+4, h(q)+8$, and so on, until we find an element smaller than q (or we hit the end of the array). Then we simply apply binary search on the interval that's guaranteed to contain q.

What is the cost of such an approach? Let $t(q)$ be the true position of q in the array (or the position of the largest element smaller than q if it is not in the set). The absolute loss of the classifier on q is then $\epsilon_q = |h(q) - t(q)|$. On the other hand, the cost of running the above algorithm starting at $h(q)$ is at most $2(\log|h(q) - t(q)|) = 2\log\epsilon_q$. If the queries q come from a distribution, then the expected cost of the algorithm is: $2\mathbb{E}_q\Big[\log\left(|h(q)-t(q)|\right)\Big] \leq 2\log\mathbb{E}_q\Big[|h(q)-t(q)|\Big] = 2\log\mathbb{E}_q[\epsilon_q]$, where the inequality follows by Jensen's inequality. This gives a trade-off between the performance of the algorithm and the absolute loss of the predictor. Moreover, since ϵ_q is trivially bounded by n, this shows that even relatively weak classifiers (those with average error of $\sqrt{n}$) this can lead to an improvement in performance.

1.3. Related work

Our work builds upon the foundational work on competitive analysis and online algorithms; for a great introduction see the book by Borodin & El-Yaniv (1998). Specifically we look at the standard caching problem, see, for example, (Motwani & Raghavan, 1995). While many variants of caching have been studied over the years, our main starting point will be the Marker algorithm by Fiat et al. (1991).

As we mentioned earlier, competitive analysis fails to distinguish between algorithms that perform well in practice, and those that perform well only in theory. Several fixes have been proposed to address these concerns, ranging from resource augmentation, where the online algorithm has a

larger cache than the offline optimum (Sleator & Tarjan, 1985), to models of real-world inputs that restrict the inputs analyzed by the algorithm, for example, insisting on locality of reference (Albers et al., 2002), or the more general Working Set model (Denning, 1968).

The idea of making assumptions on the input to prove better bounds is also common. The most popular of these is that the data arrive in a random order. This is a critical assumption in the secretary problem, and, more generally, in other streaming algorithms, see for instance the survey by McGregor (2014). While the assumption leads to algorithms that give good insight into the structure of the problem, it rarely holds true, and is often very hard to verify.

The prior work that is closest in spirit to ours looks for algorithms that optimistically assume that the input has a certain structure, but also have worst case guarantees when that fails to be the case. One such assumption is that the data are coming from a stochastic distribution and was studied in the context of online matching (Mirrokni et al., 2012) and bandit learning (Bubeck & Slivkins, 2012); both of these works provide improved guarantees if the input is stochastic but retain the worst-case guarantees. On a related note, Ailon et al. (2011) consider "self-improving" algorithms that effectively learn the input distribution, and adapt to be nearly optimal in that domain.

A more general approach was suggested by Mahdian et al. (2012), who assume the existence of an optimistic, highly competitive algorithm, and then provide a meta algorithm with a competitive ratio that interpolates between that of the worst-case algorithm and that of the optimistic one. Although this sounds similar to our approach, one of our key challenges lies in *developing* an algorithm that can use the predictions effectively. As we show, naively following the predictions can lead to disastrous results.

In other words, we do not assume anything about the data, or the availability of good algorithms that work in restricted settings. Rather, we use the oracle to implicitly classify instances into "easy" and "hard" depending on their predictability. The "easy" instances are those on which the latest machine learning technology, be it perceptrons, decision trees, SVMs, Deep Neural Networks, GANs, LSTMs, or whatever may come in the future, has small error. On these instances our goal is to take advantage of the oracle, and obtain low competitive ratios. (Importantly, our approach is completely agnostic to the inner workings of the predictor and treats it as a black box.) The "hard" instances are those with poor prediction quality where we have to rely on classical competitive analysis to obtain good results.

Very recently, two papers explored domains similar to ours. Medina & Vassilvitskii (2017) showed how to use a machine learned oracle to optimize revenue in repeated posted price auctions. Their work has a mix of offline calculations and online predictions and focuses on the specific problem of revenue optimization. Kraska et al. (2017) demonstrated empirically that introducing machine learned components to classical algorithms (in their case index lookups) can result in significant speed and storage gains. However, unlike this work, their results are experimental, and they do not provide trade-offs on the performance of their approach vis-à-vis the error of the machine learned oracle.

2. Online Algorithms with ML Advice

In this section, we introduce a general formulation to combine online algorithms with machine learning predictions, which we term *Online with Machine Learned Advice* model (OMLA). Before introducing the model, we review some basic notions from machine learning and online algorithms.

Machine learning basics. We are given a feature space $\mathcal{X}$, describing the salient characteristics of each item and a set of labels $\mathcal{Y}$. An example is a pair (x, y), where $x \in \mathcal{X}$ describes the specific features of the example, and $y \in \mathcal{Y}$ gives the corresponding label. In the binary search application, x can be thought as the query element q searched and y as its true position $t(x)$.

A hypothesis is a mapping $h : \mathcal{X} \to \mathcal{Y}$ and can be probabilistic in which case the output on $x \in \mathcal{X}$ is some probabilistically chosen $y \in \mathcal{Y}$. In binary search, $h(x)$ corresponds to the predicted position of the query.

To measure the performance of a hypothesis, we first define a loss function $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^{\geq 0}$. When the labels lie in a metric space, we define absolute loss $\ell_1(y, \hat{y}) = |y - \hat{y}|$, squared loss $\ell_2(y, \hat{y}) = (y - \hat{y})^2$, and classification loss $\ell_c(y, \hat{y}) = \mathbf{1}_{y \neq \hat{y}}$. Given a sequence of labels, $\sigma = y_1, y_2, \ldots, y_n$ and a set of predictions $\hat{\sigma} = \hat{y}_1, \ldots, \hat{y}_n$, we will abuse notation and define $\ell(\sigma, \hat{\sigma})$ as the total loss on the sequence, $\ell(\sigma, \hat{\sigma}) = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i)$.

Competitive analysis. To obtain worst-case guarantees for an online algorithm (that must make decisions when elements arrive), we compare its performance to that of an offline optimum (with the benefit of hindsight). Let σ be the input sequence of elements for a particular online decision making problem, $cost_A(\sigma)$ be the cost incurred by an offline algorithm A on this input, and $cost^\star(\sigma)$ be the cost incurred by the optimal offline algorithm. Then algorithm A is called α-competitive if $cost_A(\sigma) \leq \alpha \cdot cost^\star(\sigma)$.

Caching. The caching (or online paging) problem considers a system with two levels of memory: a slow memory of size m, and a fast memory of size k. A caching algorithm is faced with a sequence of requests for elements. If the requested element is in the fast memory, a *cache hit* occurs and the algorithm satisfies the request at no cost. If the

requested item is not in the fast memory, a *cache miss* occurs, the algorithm fetches the item from the slow memory, and places it in the fast memory before satisfying the request. If the fast memory is full, then one of the items must be evicted. The eviction strategy forms the core of the caching online algorithmic problem. The goal is to find an eviction policy that results in the fewest number of cache misses.

OMLA Definition. We first specify the input and the predictions made by the machine learned oracle. The online input consists of a set of elements $\mathcal{Z}$. For a specific input σ, its elements are denoted by $z_1, z_2, \ldots$ and its length by $|\sigma|$. Formalizing the machine learning task, we assume a feature space $\mathcal{X}$ and a label space $\mathcal{Y}$. The i-th element z_i has features $x_i \in \mathcal{X}$ and a label $y_i \in \mathcal{Y}$. In defining the framework we are not concerned with the semantics of the labels, i.e. what is the quantity that h is predicting or how it was trained—we are only interested in its performance. We define the respective total loss functions:

- *Total Classification Loss*: $\eta_c = \sum_i \ell_c(y_i, h(x_i))$,
- *Total Absolute Loss*: $\eta_1 = \sum_i \ell_1(y_i, h(x_i))$, and
- *Total Squared Loss*: $\eta_2 = \sum_i \ell_2(y_i, h(x_i))$.

Definition 1. *In the* Online with Machine Learned Advice *(OMLA) model, we are given:*

- *An input $\sigma = \{z_1, z_2, \ldots, z_{|\sigma|}\}$; each $z_i \in \mathcal{Z}$ has features $x_i \in \mathcal{X}$ and labels $y_i \in \mathcal{Y}$.*
- *A hypothesis function $h : \mathcal{X} \to \mathcal{Y}$ that predicts a label for each $x \in \mathcal{X}$.*
- *For a specific input σ and hypothesis h, we define η_c, η_1, and η_2 as described above.*

Our goal is to create online algorithms that use the advice h to achieve a good competitive ratio. Note that, in each problem instance, the algorithms depend both on the semantics of the prediction (i.e. what is being predicted) and the quality of the predictor h as measured by η. Importantly, we do not alter the definition of the competitive ratio—we expect our algorithms to work well on any sequence σ; however, the competitive ratio may depend on the total loss η.

Suppose that an algorithm $\mathcal{A}$ uses a predictor h with loss η to achieve a competitive ratio $c(\eta)$.

Definition 2. *A is α-*robust *for some function $\alpha(\eta)$, if $c(\eta)$ is non-decreasing and $c(\eta) = O(\alpha(\eta))$. In addition, we call A β-*consistent *if $c(0) = \beta$.*

We note that the above definition is with respect to the *observed* quality η of the predictor. If the instances come from a specific distribution $\mathcal{D}$, then one can also define the generalization error, $\bar{\eta} = \mathbb{E}_{\mathcal{D}}[\eta]$. As long as $\alpha(\eta)$ is concave, (e.g. $\alpha(\eta) = \log \eta$), η can be substituted with $\bar{\eta}$ by applying Jensen's inequality. (See Section 1.2 for an example.)

Definition 3. *Let $c^\star$ denote the offline optimum. $\mathcal{A}$ is γ-*competitive *if $c(\eta) \le \gamma c^\star$ for all values of η.*

The holy grail is to find 1-consistent, robust, and most competitive algorithms: they are never worse than the algorithms that do not use the ML oracle, and perform as well as the offline optimum when the oracle is perfect.

Caching with ML Advice. To instantiate the framework for the caching problem, we define the oracle, the label space of the predictions, and their semantics. The element space $\mathcal{Z}$ corresponds to the universe of elements that may be requested. The input sequence $\sigma = z_1, z_2, \cdots, z_n$ is the actual sequence of elements requested (fixed in advance and oblivious to the choices of the algorithm but unknown to it).

Each element $z_i \in \mathcal{Z}$ has corresponding feature x_i. This feature can encapsulate everything that is known about z_i at the time i, for example, the times that z_i arrived in the past. The exact choice of $\mathcal{X}$ is orthogonal to our setting, though of course richer features typically lead to smaller errors.

The machine learning task is to predict the next time a particular element will appear. The label space $\mathcal{Y}$ is thus a set of positions in the sequence, $\mathcal{Y} = \mathbb{N}^+$. Given a sequence $z_1, z_2, \ldots, z_n$, $y_i = \min_{\tau > i}\{\tau : z_\tau = z_i\}$. If the element is never seen again, we set $y_i = n + 1$. Note that y_i is completely determined by the sequence σ. We use $h(x_i)$ to denote the outcome of the prediction on an element with feature x_i; to simplify notation, we denote it by h_i.

3. Algorithms

We now delve deeper into the caching problem, and present competitive algorithms that use the ML oracle. We first show that simply trusting the oracle and following its predictions can lead to poor results even when the oracle is relatively good. This motivates the need to combine the oracle's predictions with classic tools from competitive analysis, we show how to combine the two and develop a consistent and robust algorithm.

3.1. Black-box approaches

An immediate way to use the oracle is to treat its output as the truth. This corresponds to a strategy that evicts the element that is predicted by h to appear furthest in the future. This approach, however, can lead to bad competitive ratios, even when the oracle is quite accurate on average. Consider the case when $k = 2$ and there are three elements a, b, c. The initial configuration of the cache has a, b, at which point c comes. The actual sequence has length T and, after the first element, consists of *bcbcbcbc...*. In contrast, for these elements, the oracle predicts *acbcbcbc...*, i.e. it predicts that a will reappear instantly, but is correct about all future predictions. The optimal approach in this example has one cache miss while the algorithm incurs T cache misses, because it never evicts a. Thus the algorithm has a cache miss almost every time, leading to an unbounded competitive ratio, even though the average absolute loss is $\eta_1/T = 1$. It is

tempting to "fix" this approach by evicting elements whose predicted times have passed; however, one can construct similar examples there as well (see supplementary material).

The problem is that there is an element that should be removed but the algorithm is tricked into keeping it in the cache. To deal with this in practice, most popular heuristics such as LRU (Least Recently Used) and FIFO (First In First Out) avoid evicting recent elements when some elements have been dormant for a long time. However, this imposes a strict eviction policy, and incorporating the information provided by the oracle is not straightforward.

The above examples highlight the difficulty in finding robust algorithms, i.e. those that lead to low error when the oracle error is small. We remark that turning a robust algorithm into a competitive one can be done in a black box manner, albeit suboptimally. This is shown by the following theorem whose proof is deferred to the supplementary material.

Theorem 1. For the caching problem, let A be an α-robust algorithm and B a γ-competitive algorithm. We can then create a black-box algorithm ALG that is both 9α-robust and 9γ-competitive.

While the above approach gives a black-box manner to transform consistent algorithms into consistent and competitive ones, it is far from efficient or practical. In the next section we show how to carefully modify a proposed consistent algorithm to make it more competitive.

3.2. Predictive Marker Algorithm

We now present our main technical contribution, an oracle-based adaptation of the Marker algorithm (Fiat et al., 1991) that achieves a competitive ratio of $2 \cdot \min(2 + 2\sqrt{\eta_1/OPT}, 2H_k)$ where OPT is the offline optimum on the particular instance and $H_k = 1 + 1/2 + \cdots + 1/k$ denotes the k-th Harmonic number.

Classic Marker algorithm We begin by recalling the Marker algorithm and the analysis of its performance. The algorithm runs in phases. At the beginning of each phase, all elements are unmarked. When an element arrives and is already in the cache, the element is marked. If it is not in the cache, a *random unmarked* element is evicted, the newly arrived element is placed in the cache and is marked. Once all elements are marked and a new cache miss occurs, the phase ends and we unmark all of the elements.

For the purposes of analysis, an element is called *clean* for phase r if it appears during phase r, but does not appear during phase $r-1$. In contrast, elements that also appeared in the previous phase are called *stale*. The marker algorithm has competitive ratio of $2H_k - 1$ and the analysis is tight (Achlioptas et al., 2000). We use a slightly simpler analysis that achieves competitive ratio of $2H_k$ below. The crux of the upper bound lies in two lemmas. The first relates the performance of the optimal offline algorithm to the number of clean elements L by proving that $\text{OPT} \geq 2L$ (Lemma 1). The second comes from bounding the performance of the algorithm as a function of the number of clean elements by proving that it is at most $L \cdot H_k$ in expectation (Lemma 2). For completeness, we provide the proofs of the lemmas in the supplementary material.

Lemma 1 ((Fiat et al., 1991))**.** Let L be the number of clean elements. Then the optimal algorithm suffers at least $L/2$ cache misses.

Lemma 2 ((Fiat et al., 1991))**.** Let L be the number of clean elements. Then the expected number of cache misses of the Marker algorithm is $L \cdot H_k$ when randomly tie-breaking across unmarked elements.

Predictive Marker. Delving into the analysis of the Marker algorithm, observe that it never evicts marked elements when there are unmarked elements present. This gives an upper bound of $O(k)$ on the competitive ratio for *any* tie-breaking rule that selects an unmarked element for eviction.

It is natural then to use the predictions made by the oracle for tie-breaking, specifically by evicting the element whose predicted next appearance time is furthest in the future. When the oracle is perfect (and has zero error), then stale elements never result in cache misses, and therefore, by Lemma 1, the algorithm has a competitive ratio of 2. On the other hand, by using the Marker algorithm and not blindly trusting the oracle, guarantees a worst-case ratio of $\mathcal{O}(k)$.

This is a promising direction, however an imperfect oracle may lead to high competitive ratios and perform much worse than the best offline algorithm. The problem arises when the errors of the oracle are concentrated in one phase, here the above algorithm may have a high competitive ratio. We therefore focus on creating a tie-breaking rule that gives a 2-consistent algorithm: as the oracle error goes to 0, the competitive ratio goes towards 2 while, at the same time being (approximately) competitive, i.e. keeping a worst-case $\mathcal{O}(H_k)$ competitive ratio.

To achieve this, we combine the oracle-based tie-breaking rule with the random tie-breaking rule. Suppose an element e is evicted during the phase. We construct a blame graph to understand the reason why e is evicted. There are two cases: either it was evicted when a clean element c arrived, in which case we add a directed edge from e to c, or it was evicted because a stale element s arrived, but s was previously evicted. In this case, we add a directed edge from e to s. Note that the graph is always a set of chains (paths). The total length of the chains represents the total number of evictions incurred by the algorithm during the phase, whereas the number of distinct chains represents the number of clean elements; we call the lead element in a chain, its *representative* and denote it by $\omega(r, c)$, where r is the index of the phase and c the index of the chain.

Our modification is simple—when a stale element arrives, it evicts a new element in an oracle-based manner if the corresponding clean element has slack (its chain has length less than H_k). Otherwise it evicts a random unmarked element. (In expectation this results in at most H_k elements added to any one chain during the course of the phase by the analysis of Lemma 2). This guarantees that the competitive ratio is at most $4H_k$ in expectation; we make the argument formal in Theorem 2. The crux to the analysis is the fact that the chains are disjoint, thus the interactions between eviction can be decomposed cleanly. We give a formal version of the algorithm in Algorithm 1.

3.3. Analysis

To analyze the performance of the proposed algorithm, we begin with a technical definition that captures how slowly a loss function ℓ can grow. Lemma 3 instantiates this quantity for classical losses (for a proof, see supplementary material).

Definition 4. *Let $A_T = a_1, a_2, \ldots, a_T$, be a sequence of increasing integers of length T, that is $a_1 < a_2 < \ldots < a_T$, and $B_T = b_1, b_2, \ldots, b_T$ a sequence of non-decreasing reals of length T, $b_1 \le b_2 \le \ldots \le b_T$. For a fixed loss function ℓ, we define its* spread $S_\ell : \mathbb{N}^+ \to \mathbb{R}^+$ *as:*

$$S_\ell(m) = \min\{T : \min_{A_T, B_T} \ell(A_T, B_T) \ge m\}$$

Lemma 3. For absolute loss, $\ell_1(A, B) = \sum_i |a_i - b_i|$, the spread of ℓ_1 is $S_{\ell_1}(m) \le \sqrt{4m + 1}$.
For squared loss, $\ell_2(A, B) = \sum (a_i - b_i)^2$, the spread of ℓ_2 is $S_{\ell_2}(m) \le \sqrt[3]{14m}$.

We now provide the main theorem of the paper.

Theorem 2. Suppose that the oracle has total loss η under a loss function ℓ with spread bounded by S_ℓ. If S_ℓ is concave, then the competitive ratio is at most

$$2 \cdot \min\left(1 + 2S_\ell\left(\frac{\eta}{\textsc{Opt}}\right), 2H_k\right).$$

Proof. Fix a phase of the marker algorithm, consider a particular clean element c that arrives and evicts a stale element s_1. Until s_1 arrives again, the effect of this eviction is nonexistent (as no other element is affected). When s_1 arrives, it evicts another element which we call s_2, and so on. Consider the *clean chain* consisting of $c, s_1, s_2, \ldots$. For the first H_k elements of this chain, the predicted times are in weakly decreasing order since the reason why we evicted s_i instead of s_j with $i < j$ was because the predicted time of the former was no earlier than the one of the latter (as both of them were unmarked at the time since s_j was also later evicted within the phase). However, the actual arriving times are in increasing order. Therefore, if the total loss on these elements in the chain is at most ϵ, then the number of stale elements (and the number of misses) is at most $S_\ell(\epsilon)$.

Algorithm 1 Predictive Marker with oracle-based and random tie-breaking based on clean chains

Require: Cache $\mathcal{C}$ of size k initially empty ($\mathcal{C} \leftarrow \emptyset$).
1: Initialize phase counter $r \leftarrow 1$, unmark all elements ($\mathcal{M} \leftarrow \emptyset$), and set round $i \leftarrow 1$.
2: Initialize clean element counter $\ell_r \leftarrow 0$ and clean set $\mathcal{S} \leftarrow \emptyset$.
3: Element z_i arrives, and the oracle gives a prediction h_i. Save prediction $p(z_i) \leftarrow h_i$.
4: **if** z_i results in cache hit ($z_i \in \mathcal{C}$ or $|\mathcal{C}| < k$) **then**
5: Add to cache $C \leftarrow C \cup \{z_i\}$ and go to step 26
6: **end if**
7: **if** the cache is full and all cache elements are marked ($|\mathcal{M}| = k$) **then**
8: Increase phase ($r \leftarrow r + 1$), initialize clean counter ($\ell_r \leftarrow 0$), save current cache ($\mathcal{S} \to \mathcal{C}$) as the set of elements that are possibly stale in the new phase, and unmark elements ($\mathcal{M} \leftarrow \emptyset$).
9: **end if**
10: **if** z_i is a clean element ($z_i \notin \mathcal{S}$) **then**
11: Increase number of clean elements: $\ell_r \leftarrow \ell_r + 1$.
12: Initialize size of new clean chain: $n(r, \ell_r) \leftarrow 1$.
13: Select to evict unmarked element with highest predicted time: $e = \arg\max_{z \in \mathcal{C} - \mathcal{M}} p(z)$.
14: **end if**
15: **if** z_i is a stale element ($z_i \in \mathcal{S}$) **then**
16: It is the representative of some clean chain. Let c be this clean chain: $z_i = \omega(r, c)$.
17: Increase length of clean chain $n(r, c) \leftarrow n(r, c) + 1$.
18: **if** $n(r, c) \le H_k$ **then**
19: Select to evict unmarked element with highest predicted time: $e = \arg\max_{z \in \mathcal{C} - \mathcal{M}} p(z)$.
20: **else**
21: Select to evict a random unmarked element $e \in \mathcal{C} - \mathcal{M}$.
22: **end if**
23: Update cache by evicting e: $\mathcal{C} \leftarrow \mathcal{C} \cup \{z_i\} - \{e\}$.
24: Set e as representative for the chain: $\omega(r, c) \leftarrow e$.
25: **end if**
26: Mark incoming element ($\mathcal{M} \leftarrow \mathcal{M} \cup \{z_i\}$), increase round ($i \leftarrow i + 1$), and go to step 3.

If this is higher than H_k, then the algorithm switched to random eviction which by Lemma 2 results in at most another H_k stale elements in expectation. As a result, the expected number of stale elements is never more than $2H_k$ and is less than $S_\ell(\epsilon)$ when this quantity is less than H_k; it is therefore upper bounded by $\min(2 \cdot S_\ell(\epsilon), 2H_k)$.

Let L be the number of clean elements (and therefore also chains). Since both S_ℓ and the minimum operator are concave functions, the way to maximize the number of stale elements in each chain is to apportion the total error, η,

equally across all of the chains. Thus there are L chains with error η/L each. The total number of stale elements is then: $L \cdot \min(2 \cdot S_\ell(\eta/L), 2H_k)$. By Lemma 1, $L/2 \leq \text{OPT}$, which implies the result since also trivially $\text{OPT} \leq L$. □

We now specialize the results to absolute and squared losses.

Corollary 1. The competitive ratio of Algorithm 1 when the oracle has ℓ_1 error η_1 is at most $\min\left(2 + 2\sqrt{4 \cdot \eta_1/\text{OPT} + 1}, 4H_k\right)$.

Corollary 2. The competitive ratio of Algorithm 1 when the oracle has squared loss η_2, is at most $\min\left(2 + 2\sqrt[3]{14 \cdot \eta_2/\text{OPT}}, 4H_k\right)$.

3.4. Discussion and Extensions

We have shown how to tie the loss of the machine learned oracle h to the performance of the Predictive Marker, and gave a bound on the interplay of the two. We explore additional extensions to the algorithm below, giving a general trade-off between its robustness and competitiveness, as well as a tighter analysis on its performance. Finally, we show how to view the LRU algorithm as a variant of Predictive Marker with a specific, easy to compute objective function.

Robustness vs. Competitiveness. One parameter in Algorithm 1 is the length of the chain when the algorithm switches from following the oracle to random unmarked evictions. If the switch occurs at length γH_k, this provides a trade-off between competitiveness and robustness.

Theorem 3. Suppose that, for some $\gamma > 0$, the algorithm uses γH_k as switching point, the oracle has total loss η under a loss function ℓ with spread bounded by S_ℓ. If S_ℓ is concave, then the competitive ratio is at most

$$2 \cdot \min\left(1 + \frac{1+\gamma}{\gamma} S_\ell\left(\frac{\eta}{\text{OPT}}\right), (1+\gamma)H_k\right).$$

Note that setting γ close to 0 makes the algorithm more conservative (switching to random evictions earlier), and thus reduces the competitive ratio when the oracle error is large. On the other hand, setting γ high has the algorithm trusting the oracle more, and reduces the competitive ratio when the oracle error is small.

Tighter Analysis. Standard loss functions like absolute and squared loss are defined on a per element basis. On the other hand, we can get a tighter bound on the performance of Predictive Marker, if we compare the *sequence* generated by the oracle with the ground truth.

Let (e, i) be the pair that corresponds to the i-th arrival of element e. Create the sequence A_T by putting these pairs in increasing order of their true arrival time and B_T by putting them in increasing order of their predicted arrival time. The *edit distance*, ℓ_{ed}, between these two sequences precisely captures the performance of Predictive Marker.

Theorem 4. The competitive ratio of Algorithm 1 when the oracle has ℓ_{ed} error η_{ed} is at most $\min\left(3 + 2\frac{\eta_{ed}}{\text{OPT}}, 4H_k\right)$

Proof. For any clean chain, the first $m \leq H_k$ stale elements are in inverse order in A_T and B_T; else they would not be evicted. Hence these elements are certainly misplaced in the edit distance metric and contribute error of $m - 1$. The rest of the proof follows the same steps as in Theorem 2. □

Unifying Framework for Caching. We remark that one can express the popular Least Recently Used (LRU) algorithm for caching in the framework above. Suppose for an element that appears at time i we predict its next appearance at time $-i$. Then the element that is predicted to appear furthest in the future is exactly the one that has appeared least recently. PredictiveMarker with these predictions exactly simulates LRU when the switching thrshold is k. The reason is that just like Marker, such an implementation of LRU never removes a marked element (that appeared more recently) when an ummarked element (that appeared earlier) is present. This implies that we can make LRU more robust by combining it with random eviction in case there are many errors accumulated in some phase.

Similarly, the Classic Marker algorithm can be written in the framework with any predictor and switching threshold of 0 (implyihgs that we immediately move to random eviction).

4. Experiments

In this section we evaluate our approach on real world datasets, empirically demonstrate its dependence on the errors in the oracle, and compare it to standard baselines, like LRU and Marker.

Datasets and Metrics. We explore two datasets from different domains to show the wide applicability of our approach.

- `BK` is data extracted from BrightKite, a now defunct social network. We consider sequences of checkins, and extract the top 100 users with the longest non-trivial check in sequences—those where the optimum policy has at least 50 misses. This dataset is publicly available at (Cho et al., 2011; Bri). Each user sequence represents an instance of the caching problem.
- `Citi` is data extracted from CitiBike, a popular bike sharing platform operating in New York City. We consider citi bike trip histories, and extract stations corresponding to starting points of each trip. We create 12 sequences, one for each month of 2017 for this dataset. We consider only the first 25,000 events in each file. The dataset is publicly available at (Cit).

We give additional statistics about each datasets in Table 1.

Our main metric for evaluation will be the *competitive ratio* of the algorithm, defined as the number of misses incurred

Dataset	Num Sequences	Sequence Length	Unique Elements
`BK`	100	2,101	67– 800
`Citi`	24	25,000	593 – 719

Table 1. Number of sequences; sequence length; min and max number of elements for each dataset.

divided by the optimum number of misses.

Predictions. We run experiments with both synthetic predictions to showcase the sensitivity of our methods to learning errors, and with preditions using an off the shelf classifier, published previously (Anderson et al., 2014).

- **Synthetic Predictions.** For each element, we first compute the true next arrival time $y(t)$, setting it to $n+1$ if it does not appear in the future. To simulate the performance of an ML system, we set $h(t) = y(t) + \epsilon$, where ϵ is drawn i.i.d. from a lognormal distribution with mean parameter 0 and standard deviation σ. We chose the lognormal distribution of errors to demonstrate the sensitivity to rare but large failures of the learning algorithm. Finally, since we only compare the relative predicted times for each method, adding a bias term to the predictor would not change the results.
- **PLECO Predictions.** In their work, Anderson et al. (2014) developed a simple framework to model repeat consumption, and published the parameters of their PLECO (Power Law with Exponential Cut Off) model for the BrightKite dataset. While their work focused on predicting the relative probabilities of each element (re)appearing in the subsequent time step, we modify it to predict the next time an element will appear. Specifically, we set $h(t) = t + 1/p(t)$, where $p(t)$ represents the probability that element that appeared at time t will re-appear at time $t + 1$.

Algorithms. We use multiple algorithms for evaluation.

- **LRU** is the Least Recently Used policy that is wildly successful in practice.
- **Marker** is the classical algorithm of Fiat et al. (1991).
- **PredictiveMarker** is the algorithm we develop in this work. We set the switching cost to k, and therefore never switch to random evictions.
- **Blind Oracle** is the algorithm of Section 3.1, evicting the element predicted to appear furthest in the future.

4.1. Results

We set $k = 10$, and summarize the synthetic results on the `BK` dataset in Figure 1. Observe that the performance of Predictive Marker is consistently better than LRU and standard Marker, and degrades slowly as the average error increases, as captured by the theoretical analysis. Second, we empirically verify that blindly trusting the oracle works well when the error is very low, but quickly becomes incredibly costly.

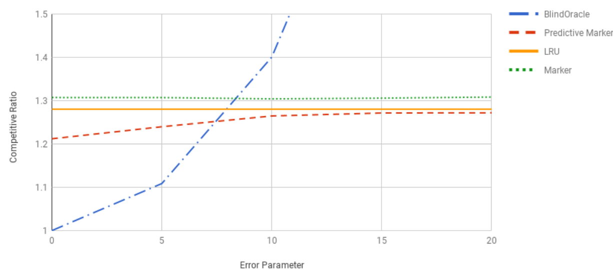

Figure 1. Ratio of average number of evictions as compared to optimum for varying levels of oracle error.

Algorithm	`BK`	`Citi`
Blind Oracle	2.049	2.023
LRU	1.280	1.859
Marker	1.310	1.869
Predictive Marker	1.266	1.810

Table 2. Competitive Ratios using PLECO model on both datasets.

The results using the PLECO predictor are shown in Table 2, where we keep $k = 10$ for `BK` and set $k = 100$ for `Citi`; the ranking of the methods is not sensitive to cache size. We can again see that the Predictive Marker algorithm outperforms all othersn and is 2.5% better than the next best method, LRU. While the gains appear modest, they are statistically significant at $p < 0.001$. Moreover, the off-the-shelf PLECO model was not tuned or optimized for predicting the *next* appearance of each element.

5. Conclusion

In this work, we introduce the study of online algorithms with the aid of machine learned oracles. This combines the empirical success of machine learning with the rigorous guarantees of online algorithms. We model the setting for the classical caching problem and give an oracle-based algorithm whose competitive ratio is directly tied to the accuracy of the machine learned oracle.

Our work opens up two avenues for future work. On the theoretical side, it would be interesting to see similar oracle-based algorithms for other online settings. On the practical side, our caching algorithm shows how we can use machine learning in a safe way, avoiding problems caused by rare wildly inaccurate predictions. At the same time, our experimental results show that even with simple predictors, our algorithm provides improvement compared to LRU. In essence, we have reduced the worst case performance of the caching problem to that of finding a good (on average) predictor. This opens up the door for practical algorithms that yield provably good performance without being tailored towards the worst-case or specific distributional assumptions.

Acknowledgements

The authors thank Andrés Muñoz-Medina and Éva Tardos for valuable discussions and an anonymous reviewer for pointing towards the direction of Theorem 1. The first author was supported under NSF grant CCF-1563714. Part of the work was done while the author was interning at Google.

References

Brightkite data. http://snap.stanford.edu/data/loc-brightkite.html.

Citibike system data. http://https://www.citibikenyc.com/system-data.

Achlioptas, D., Chrobak, M., and Noga, J. Competitive analysis of randomized paging algorithms. *Theor. Comput. Sci.*, 234(1-2):203–218, 2000. doi: 10.1016/S0304-3975(98)00116-9. URL https://doi.org/10.1016/S0304-3975(98)00116-9.

Ailon, N., Chazelle, B., Clarkson, K. L., Liu, D., Mulzer, W., and Seshadhri, C. Self-improving algorithms. *SIAM J. Comput.*, 40(2):350–375, 2011. doi: 10.1137/090766437. URL https://doi.org/10.1137/090766437.

Albers, S., Favrholdt, L. M., and Giel, O. On paging with locality of reference. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pp. 258–267, New York, NY, USA, 2002. ACM. ISBN 1-58113-495-9. doi: 10.1145/509907.509949. URL http://doi.acm.org/10.1145/509907.509949.

Anderson, A., Kumar, R., Tomkins, A., and Vassilvitskii, S. The dynamics of repeat consumption. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pp. 419–430, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2744-2. doi: 10.1145/2566486.2568018. URL http://doi.acm.org/10.1145/2566486.2568018.

Borodin, A. and El-Yaniv, R. *Online Computation and Competitive Analysis*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-56392-5.

Bubeck, S. and Slivkins, A. The best of both worlds: Stochastic and adversarial bandits. In *COLT 2012 - The 25th Annual Conference on Learning Theory, June 25-27, 2012, Edinburgh, Scotland*, pp. 42.1–42.23, 2012. URL http://www.jmlr.org/proceedings/papers/v23/bubeck12b/bubeck12b.pdf.

Cho, E., Myers, S. A., and Leskovec, J. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pp. 1082–1090, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0813-7. doi: 10.1145/2020408.2020579. URL http://doi.acm.org/10.1145/2020408.2020579.

Denning, P. J. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968. ISSN 0001-0782. doi: 10.1145/363095.363141. URL http://doi.acm.org/10.1145/363095.363141.

Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. Competitive paging algorithms. *J. Algorithms*, 12(4):685–699, December 1991. ISSN 0196-6774. doi: 10.1016/0196-6774(91)90041-V. URL http://dx.doi.org/10.1016/0196-6774(91)90041-V.

Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. The case for learned index structures. 2017. URL https://arxiv.org/abs/1712.01208.

Mahdian, M., Nazerzadeh, H., and Saberi, A. Online optimization with uncertain information. *ACM Trans. Algorithms*, 8(1):2:1–2:29, 2012. doi: 10.1145/2071379.2071381. URL http://doi.acm.org/10.1145/2071379.2071381.

McGregor, A. Graph stream algorithms: A survey. *SIGMOD Rec.*, 43(1):9–20, May 2014. ISSN 0163-5808. doi: 10.1145/2627692.2627694. URL http://doi.acm.org/10.1145/2627692.2627694.

Medina, A. M. and Vassilvitskii, S. Revenue optimization with approximate bid predictions. *CoRR*, abs/1706.04732, 2017. URL http://arxiv.org/abs/1706.04732.

Mirrokni, V. S., Gharan, S. O., and Zadimoghaddam, M. Simultaneous approximations for adversarial and stochastic online budgeted allocation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pp. 1690–1701, 2012. URL http://portal.acm.org/citation.cfm?id=2095250&CFID=63838676&CFTOKEN=79617016.

Motwani, R. and Raghavan, P. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995. ISBN 0-521-47465-5, 9780521474658.

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., and Dennison, D. Hidden technical debt in machine learning systems. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pp. 2503–2511, Cambridge, MA, USA, 2015. MIT Press. URL http://dl.acm.org/citation.cfm?id=2969442.2969519.

Sleator, D. D. and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28 (2):202–208, February 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL http://doi.acm.org/10.1145/2786.2793.

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014. URL http://arxiv.org/abs/1312.6199.