Competitive caching with machine learned advice

Thodoris Lykouris*  Sergei Vassilvitskii†

Abstract

Traditional online algorithms encapsulate decision making under uncertainty, and give ways to hedge against all possible future events, while guaranteeing a nearly optimal solution, as compared to an offline optimum. On the other hand, machine learning algorithms are in the business of extrapolating patterns found in the data to predict the future, and usually come with strong guarantees on the expected generalization error.

In this work 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 when using simple off-the-shelf predictions.

*Cornell University, teddlyk@cs.cornell.edu. Work supported under NSF grant CCF-1563714. Part of the work was done while the author was interning at Google.
†Google Research, sergeiv@google.com.
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 [SHG+15]. 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 in 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 [SZS+14], 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 future inputs. 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 translates 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, a series of requests arrives 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 incurs 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 what is observed in practice.

In contrast, we show how to use machine learned predictions to achieve a competitive ratio of $2 + O(\min(\sqrt{\eta/\text{opt}}, \log k))$, when using a hypothesis with total absolute loss $\eta$ (see Section 3 for a precise statement of results). Thus when the predictions are accurate (small $\eta$), our approach circumvents the worst case lower bounds. On the other hand, even when the oracle is inaccurate (large $\eta$), the performance degrades gracefully to almost match the worst case bound.

1.1 Our contribution

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 of the oracle, $\eta$, and not the distribution of the error.
- Are robust: a better oracle (one with lower $\eta$) results in a smaller competitive ratio
- Are worst-case competitive: no matter the performance of the oracle on the particular instance, the algorithm behaves 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 \cite{FKL+91} 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 a simple 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 applying 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 \cite{KBC+17} 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:

$$2\mathbb{E}_q \left[ \log (|h(q) - t(q)|) \right] \leq 2 \log \mathbb{E}_q [ |h(q) - t(q)| ] = 2 \log \mathbb{E}_q [\epsilon_q],$$

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 $\epsilon_q$ 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 and El-Yaniv [BEY98]. Specifically, we look at the standard caching problem, see, for example, [MR95]. While many variants of caching have been studied over the years, our main starting point will be the Marker algorithm by Fiat et al. [FKL+91].

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 [ST85], to models of real-world inputs that restrict the inputs analyzed by the algorithm, for example, insisting on locality of reference [AFG02], or the more general Working Set model [Den68].

The idea of making assumptions on the nature of the input to prove better bounds is common in the literature. 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 [McG14]. 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.

Another common assumption on the structure of the input gives rise to Smoothed Analysis, introduced in a pioneering work by Spielman and Teng [ST04] explaining the practical efficiency of the Simplex method. Here, they assume that any worst case instance is perturbed slightly before being passed to the algorithm; the idea is that this perturbation may be due to measurement error, or some other noise inherent in the data. The goal then is to show that the worst case inputs are brittle, and do not survive addition of random noise. Since its introduction this method has been used to explain the unusual effectiveness of many practical algorithms such as ICP [AV06], Lloyd’s method [AMR11], and local search [ERV16], in the face of exponential worst case bounds.

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 [MGZ12] and bandit learning [BS12]; both of these works provide improved guarantees if the input is stochastic but retain the worst-case guarantees otherwise. Taking this idea further, Ailon et al. [ACC+11] 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. [MNS12], 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 else 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 where the prediction quality is poor, and we have to rely more on classical competitive analysis to obtain good results.

A previous line of work has also considered the benefit of enhancing online algorithms with oracle advice (see [BFK+16] for a recent survey). This setting assumes access to an infallible oracle and studies the amount of information that is needed to achieve desired competitive ratio guarantees. Our work differs in two major regards. First, we do not assume that the oracle is perfect, as that is rarely the case in machine learning scenarios. Second, we study the trade-off between oracle error and the competitive ratio, rather than focusing on the number of perfect predictions necessary.

Another avenue of research close to our setting asks what happens if the algorithm cannot view the whole input, but must rely on a sample of the input to make its choices. Introduced in the seminal work of Cole and Roughgarden [CR14], this notion of Learning from Samples, can be viewed as first designing a good prediction function, $h$, and then using it in the algorithms. Indeed, some of the follow up work [MR16, BRS17] proves tight bounds on precisely how many samples are necessary to achieve good approximation guarantees. In contrast, we assume that the online algorithm is given access to a machine learned oracle, but does not know any details of its inner workings—we know neither the average performance of the oracle, nor the distribution of the errors.

Very recently, two papers explored domains similar to ours. Medina and Vassilvitskii [MV17] 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. [KBC+17] 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 Machine Learned Advice

In this section, we introduce a general framework for combining 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.
2.1 Preliminaries

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

A hypothesis is a mapping \( h : X \rightarrow Y \) and can be probabilistic in which case the output on \( x \in X \) is some probabilistically chosen \( y \in Y \). 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 \( \ell : Y \times Y \rightarrow \mathbb{R}^{\geq 0} \). 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 \( \ell_c(y, \hat{y}) = 1_{y \neq \hat{y}} \). 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 as each element arrives), we compare its performance to that of an offline optimum (that has the benefit of hindsight). Let \( \sigma \) be the input sequence of elements for a particular online decision making problem, \( \text{cost}_A(\sigma) \) be the cost incurred by an offline algorithm \( A \) on this input, and \( \text{cost}^*(\sigma) \) be the cost incurred by the optimal offline algorithm. Then algorithm \( A \) is called \( \alpha \)-competitive if

\[
\text{cost}_A(\sigma) \leq \alpha \cdot \text{cost}^*(\sigma).
\]

The Caching Problem  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 can satisfy 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 problem. The goal is to find an eviction policy that results in the fewest number of cache misses.

It is well known that the optimal offline algorithm at time \( t \) evicts the element from the cache that will arrive the furthest in the future; this is typically referred in the literature as Bélády’s optimal replacement paging algorithm [Bel66]. On the other hand, without the benefit of foresight, any deterministic caching algorithm achieves a competitive ratio of \( \Omega(k) \), and any randomized caching algorithm achieves a competitive ratio of \( \Omega(\log k) \) [MR95].

2.2 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 \( Z \). For a specific input \( \sigma \), 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 \( X \) and a label space \( Y \). The \( i \)-th element \( z_i \) has features \( x_i \in X \) and a label \( y_i \in Y \).
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 \( \sigma \) and hypothesis \( h \), we define \( \eta_c, \eta_1, \) and \( \eta_2 \) as described above.

Our goal is to create online algorithms that can 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 \( \eta \).

Importantly, we do not alter the definition of the competitive ratio—we expect our algorithms to work well on any sequence \( \sigma \); however, the competitive ratio may depend on the total loss \( \eta \).

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

**Definition 2.** \( A \) is \( \alpha \)-robust for some function \( \alpha(\eta) \), if \( c(\eta) \) is non-decreasing and \( c(\eta) = O(\alpha(\eta)) \).

**Definition 3.** \( A \) is \( \beta \)-consistent if \( c(0) = \beta \).

We note that both of the above definitions are with respect to the ex post quality \( \eta \) of the predictor.

If the problem instances are assumed to 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 (for instance, \( \alpha(\eta) = \log \eta \)), \( \eta \) can be replaced with \( \bar{\eta} \) by applying Jensen’s inequality. (See Section 1.2 for an example.)

**Definition 4.** Let \( c^* \) denote the offline optimum. \( A \) is \( \gamma \)-competitive if \( c(\eta) \leq \gamma c^* \) for all values of \( \eta \).

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.

### 2.3 Caching with ML Advice

To instantiate the framework for the caching problem, we need to 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 \( \sigma = z_1, z_2, \ldots, z_\sigma \) is the actual sequence of elements that are 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 features \( x_i \). These 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 is going to appear. As such, the label space $\mathcal{Y}$ is 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 $\sigma$.

We use $h(x_i)$ to denote the outcome of the prediction on an element with features $x_i$. To simplify notation, we denote this quantity 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 integrate 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 truth. This corresponds to a strategy that evicts the element that is predicted by $h$ to appear furthest into 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 $c$ arrives. The actual sequence has length $T$ and, after the first element, consists of $bcbcbcbcb...$. In contrast, for these elements, the oracle predicts $acbcbcbcb...$, 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 incurs a cache miss almost every time, leading to an unbounded competitive ratio, even though the average absolute loss is constant: $\eta \equiv 1 / T = 3$.

It is tempting to “fix” this issue by evicting elements whose predicted times have passed, but follow the same idea otherwise. Formally, let $h(j, t)$ denote the last prediction about $z_j$ at or prior to time $t$. At time $t$ this “fixed” approach evicts an arbitrary item from the set $S_t = \{ j : h(j, t) < t \}$ if $S_t \neq \emptyset$ and $\arg \max_{z_i \in \text{Cache}(t)} h(i, t)$ otherwise. We show that the competitive ratio of this algorithm is also unbounded even when the average absolute loss is constant. Assume a cache of size $k = 3$ and four elements $a, b, c, d$. The initial configuration is $a, b, c$, when element $d$ arrives. Let the total length of the sequence be $T$. The true sequence has element $b$ at the last time $T$, element $c$ at times $2^r + 1$ where $r$ is odd, $d$ at times $2^r + 1$ where $r$ is even, and $a$ at all other times. The oracle predicts the next appearance of $a$ and $b$ correctly but is incorrect on elements $c$ and $d$ and always predicts their next appearance time as $T + 1$. The optimal algorithm evicts $b$, and has two cache misses (when $d$ arrives for the first time, and when $b$ arrives at time $T$). On the other hand, the described algorithm keeps $a, b$ always in the cache and incurs a cache miss for each appearance of $c$, and $d$, a total of $\log(T)$ misses. This is despite the fact that the average absolute loss is again a constant: $\eta / T = 3$ as the errors form a geometric series.

The common problem in both examples 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 adding additional 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.

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

**Proof.** We proceed by simulating $A$ and $B$ in parallel on the dataset, and maintaining the cache state and the number of misses incurred by each. Our algorithm switches between following the strategy of $A$ and the strategy of $B$. Let $c_t(A)$ and $c_t(B)$ denote the cost (number of misses) of $A$ and $B$ up to time $t$. Without loss of generality, let $ALG$ begin by following strategy of $A$; it will do so until a time $t$ where $c_t(A) = 2 \cdot c_t(B)$. At this point $ALG$ switches to following the eviction strategy of $B$, doing so until the simulated cost of $B$ is double that of $A$: a time $t'$ with $c_{t'}(B) = 2 \cdot c_{t'}(A)$. At this point it switches back to following eviction strategy of $A$, and so on. When $ALG$ switches from $A$ to $B$, the elements that $A$ has in cache may not be the same as those that $B$ has in the cache. In this case, it needs to reconcile the two. However, this can be done lazily (at the cost of an extra cache miss for every element that needs to be reconciled). To prove the bound on the performance of the algorithm, we need to show that $c_t(ALG) \leq 9 \cdot \min(c_t(A), c_t(B))$ for all $t$. We decompose the cost incurred by $ALG$ into that due to following the different algorithms, which we denote by $f_t(ALG)$, and that due to reconciling caches, $r_t(ALG)$.

We prove a bound on the following cost $f_t$ by induction on the number of switches. Without loss of generality, suppose that at time $t$, $ALG$ switched from $A$ to $B$, and at time $t'$ it switches from $B$ back to $A$. By induction, suppose that $f_t(ALG) \leq 3 \min(c_t(A), c_t(B)) = 3c_t(B)$, where the equality follows since $ALG$ switched from $A$ to $B$ at time $t$. In both cases, assume that caches are instantly reconciled. Then:

$$f_{t'}(ALG) = f_t(ALG) + (c_{t'}(B) - c_t(B))$$

$$= f_t(ALG) + 2c_{t'}(A) - 1/2c_t(A)$$

$$\leq 3c_t(B) + 2(c_{t'}(A) - c_t(A)) + 3/2 \cdot c_t(A)$$

$$= 3c_t(A) + 2(c_{t'}(A) - c_t(A))$$

$$\leq 3c_{t'}(A)$$

$$= 3 \min(c_{t'}(A), c_{t'}(B))$$

What is left is to bound the following cost for the time since the last switch. Let $s$ denote the time of the last switch and, assume without loss of generality that it was done from $A$ to $B$. Let $s'$ denote the last time step. By the previous set of inequalities (changing the second equation to inequality) and the fact that the algorithm never switched back to $A$ after $s$, it holds that $f_{s'}(ALG) \leq 3c_{s'}(A) \leq 6 \min(c_{s'}(A), c_{s'}(B))$.

To bound the reconciliation cost, assume the switch at time $t$ is from $A$ to $B$. We charge the reconciliation of each element in $B \setminus A$ to the cache miss when the element was last evicted by $A$. Therefore the overall reconciliation cost is bounded by $r_t(ALG) \leq c_t(A) + c_t(B) \leq 3 \min(c_t(A), c_t(B))$.

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 [FKL+91] that achieves a competitive ratio of $2 \cdot \min(1 + O(\sqrt{\eta_1/OPT}), 2H_k)$ where OPT is the offline optimum on the particular instance and $H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$ 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 [ACN00]. 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 these lemmas in Appendix A.

**Lemma 1** ([FKL+91]). Let $L$ be the number of clean elements. Then the optimal algorithm suffers at least $L/2$ cache misses.

**Lemma 2** ([FKL+91]). 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 $O(k)$.

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 $O(H_k)$ 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. 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 every chain, the representative and denote it by \( \omega(r,c) \), where \( r \) is the index of the phase and \( c \) the index of the chain in the phase.

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 evictions can be decomposed cleanly. We give a formal version of the algorithm in Algorithm 1.

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

Require: Cache \( C \) of size \( k \) initially empty (\( C \leftarrow \emptyset \)).

1: Initialize phase counter \( r \leftarrow 1 \), unmark all elements (\( M \leftarrow \emptyset \)), and set round \( i \leftarrow 1 \).
2: Initialize clean element counter \( \ell_r \leftarrow 0 \) and clean set \( S \leftarrow \emptyset \).
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 (\( z_i \in C \) or \( |C| < k \)) 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 (\( |M| = k \)) then
8: Increase phase (\( r \leftarrow r + 1 \)), initialize clean counter (\( \ell_r \leftarrow 0 \)), save current cache (\( S \rightarrow C \)) as the set of elements that are possibly stale in the new phase, and unmark elements (\( M \leftarrow \emptyset \)).
9: end if
10: if \( z_i \) is a clean element (\( z_i \notin S \)) 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: \( e = \arg \max_{z \in C - M} p(z) \).
14: end if
15: if \( z_i \) is a stale element (\( z_i \in S \)) 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 the clean chain \( n(r,c) \leftarrow n(r,c) + 1 \).
18: if \( n(r,c) \leq H_k \) then
19: Select to evict unmarked element with highest predicted time: \( e = \arg \max_{z \in C - M} p(z) \).
20: else
21: Select to evict a random unmarked element \( e \in C - M \).
22: end if
23: Update cache by evicting \( e \): \( C \leftarrow C \cup \{z_i\} - \{e\} \).
24: Set \( e \) as representative for the chain: \( \omega(r,c) \leftarrow e \).
25: end if
26: Mark incoming element (\( M \leftarrow M \cup \{z_i\} \)), increase round (\( i \leftarrow i + 1 \)), and go to step 3.
3.3 Analysis

In order to analyze the performance of the proposed algorithm, we begin with a technical definition that captures how slowly a loss function $\ell$ can grow.

**Definition 5.** 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 $T$, $b_1 \leq b_2 \leq \ldots \leq b_T$. For a fixed loss function $\ell$, we define its spread $S_\ell : \mathbb{N}^+ \to \mathbb{R}^+$ as:

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

The following Lemma is proved in the Appendix:

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

We now provide the main theorem of the paper.

**Theorem 2.** Suppose that the oracle has total loss $\eta$ under a loss function $\ell$ with spread bounded by $S_\ell$. If $S_\ell$ is concave, then the competitive ratio is at most

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

**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 non-existent (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 $\epsilon$, then the number of stale elements (and the number of misses) is at most $S_{\ell}(\epsilon)$. 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 $\min(2 : S_{\ell}(\epsilon), 2H_k)$.

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

We now specialize the results for the absolute and squared losses.

**Corollary 1.** The competitive ratio of Algorithm 1 when the oracle has $\ell_1$ error $\eta_1$ is at most

$$\min \left(2 + 2\sqrt{4\eta_1/\text{OPT}} + 1, 4H_k\right).$$

**Corollary 2.** The competitive ratio of Algorithm 1 when the oracle has squared loss $\eta_2$, is at most

$$\min \left(2 + 2\sqrt{14\eta_2/\text{OPT}}, 4H_k\right).$$
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 more general trade-off between its robustness and competitiveness, as well as giving 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 of the free parameters in Algorithm 1 is on the length of the chain when the algorithm switches from following the oracle to random unmarked evictions. If the switch occurs after chains grow to $\gamma H_k$ in length, this provides a trade-off between competitiveness and robustness.

**Theorem 3.** Suppose that, for some $\gamma > 0$, the algorithm uses $\gamma H_k$ as switching point, the oracle has total loss $\eta$ under a loss function $\ell$ with spread bounded by $S_\ell$. If $S_\ell$ 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 $\gamma$ 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 $\gamma$ 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.

Formally, 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, $\ell_{ed}$, between these two sequences precisely captures the performance of Predictive Marker.

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

**Proof.** Note that 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. Therefore these elements are certainly misplaced in the edit distance metric and contribute an error of $m - 1$. The rest of the proof follows the same steps as the one of 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 threshold 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 unmarked 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 a switching threshold of 0 (which implies that we always immediately move to random eviction).

In the subsequent section, we present empirical results to show how Predictive Marker compares to these algorithms.

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.

Datasets and Metrics We consider two datasets taken from different domains to demonstrate 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 would have at least 50 misses. This dataset is publicly available at [CML11, Bri]. Each of the user sequences 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 the New York City dataset. We consider only the first 25,000 events in each file. This data is publicly available at [Cit].

We give some additional statistics about each datasets in Table 1.

| 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.

Our main metric for evaluation will be the competitive ratio of the algorithm, defined as the number of misses incurred by the particular strategy 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 predictions using an off the shelf classifier, published previously [AKTV14].

• **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 \( \epsilon \) is drawn i.i.d. from a lognormal distribution with mean parameter 0 and standard deviation \( \sigma \). We chose the lognormal distribution of errors to
showcase the effect of rare but large failures of the learning algorithm. Finally, observe that 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. [AKTV14] 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 consider multiple algorithms for evaluation.

- **LRU** is the Least Recently Used policy that is wildly successful in practice.
- **Marker** is the classical Marker algorithm due to Fiat et al. [FKL+91].
- **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 described in Section 3.1 which evicts 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 following the oracle works well when the error is very low, but quickly becomes incredibly costly.

The results using the PLECO predictor are shown in Table 2, where we keep \( k = 10 \) for the BK dataset and set \( k = 100 \) for Citi; we note that the ranking of the methods is not sensitive to the cache size, \( k \). We can again see that the Predictive Marker algorithm outperforms all others, and is 2.5% better than the next best method, LRU. While the gains appear modest, we note they are
| Algorithm         | Competitive Ratio on BK | Competitive Ratio on 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 Ratio using PLECO model.

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.

In that regard, the large difference in performance between using the predictor directly (Blind Oracle) and using it in combination with Marker (Predictive Marker) speaks to the power of the algorithmic method. By considering only the straightforward use of the predictor in the Blind Oracle setting, one may deem the ML approach not powerful enough for this application; what we show is that a more judicious use of the same model can result in tangible and statistically significant gains.

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 such as the $k$-server problem. 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 an 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 need not be tailored towards the worst-case or specific distributional assumptions, but still yield provably good performance.

Acknowledgements

The authors would like to thank Andrés Muñoz-Medina and Éva Tardos for valuable discussions as well as an anonymous reviewer for pointing towards the direction of Theorem 1.

References

[ACC+11] Nir Ailon, Bernard Chazelle, Kenneth L. Clarkson, Ding Liu, Wolfgang Mulzer, and C. Seshadhri. Self-improving algorithms. *SIAM J. Comput.*, 40(2):350–375, 2011.

[ACN00] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theor. Comput. Sci.*, 234(1-2):203–218, 2000.
[AFG02] Susanne Albers, Lene M. Favrholdt, and Oliver Giel. On paging with locality of reference. In Proceedings of the Thirty-fourth Annual ACM Symposium on Theory of Computing, STOC ’02, pages 258–267, New York, NY, USA, 2002. ACM.

[AKTV14] Ashton Anderson, Ravi Kumar, Andrew Tomkins, and Sergei Vassilvitskii. The dynamics of repeat consumption. In Proceedings of the 23rd International Conference on World Wide Web, WWW ’14, pages 419–430, New York, NY, USA, 2014. ACM.

[AMR11] David Arthur, Bodo Manthey, and Heiko Röglin. Smoothed analysis of the k-means method. J. ACM, 58(5):19:1–19:31, 2011.

[AV06] David Arthur and Sergei Vassilvitskii. Worst-case and smoothed analysis of the ICP algorithm, with an application to the k-means method. In 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings, pages 153–164, 2006.

[Bel66] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Syst. J., 5(2):78–101, June 1966.

[BEY98] Allan Borodin and Ran El-Yaniv. Online Computation and Competitive Analysis. Cambridge University Press, New York, NY, USA, 1998.

[BFK+16] Joan Boyar, Lene M. Favrholdt, Christian Kudahl, Kim S. Larsen, and Jesper W. Mikkelsen. Online algorithms with advice: A survey. SIGACT News, 47(3):93–129, August 2016.

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

[BRS17] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. The limitations of optimization from samples. In STOC, 2017.

[BS12] Sébastien Bubeck and Aleksandrs Slivkins. 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, pages 42.1–42.23, 2012.

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

[CML11] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. 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, pages 1082–1090, New York, NY, USA, 2011. ACM.

[CR14] Richard Cole and Tim Roughgarden. The sample complexity of revenue maximization. In Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014, pages 243–252, 2014.

[Den68] Peter J. Denning. The working set model for program behavior. Commun. ACM, 11(5):323–333, May 1968.

[ERV16] Matthias Englert, Heiko Röglin, and Berthold Vöcking. Smoothed analysis of the 2-opt algorithm for the general TSP. ACM Trans. Algorithms, 13(1):10:1–10:15, 2016.

[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.
A Marker Algorithm Proofs

In this section, we provide the proofs of two crucial lemmas in the classical Marker algorithm analysis. The proofs are due to Fiat et al. [FKL+91] and are stated for completeness.

**Lemma [1] restated:** Let \( L \) be the number of clean elements. Then the optimal algorithm suffers at least \( \frac{L}{2} \) cache misses.

**Proof.** We denote by \( \ell_r \) the number of clean elements at phase \( r \). Let \( d_r^I \) be the number of elements that are in the cache of the optimal solution but not in the cache of the algorithm’s solution at the beginning of phase \( r \) and let \( d_r^F \) be the number of elements that are in this set at the end of the phase. Then, the number of cache misses of the optimal solution are at least the difference \( \ell_r - d_r^I \).
(as any clean element that was not in the initial complement would cause one cache miss) and also at least $d_r^F$ (since these many elements did not arise during the phase and the offline algorithm has for sure incurred these many misses). Hence, for each $r$, we know that:

$$\#\text{cache misses in opt during phase } r \geq \max(\ell_r - d_{r}^{l}, d_{r}^{F}) \geq \frac{\ell_r - d_{r}^{l} + d_{r}^{F}}{2}.$$  

Telescoping and using that $d_0^l = 0$ (since both start with the same cache elements) concludes the lemma.

**Lemma 2** restated: 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.

**Proof.** We again denote by $\ell_r$ the number of clean elements at phase $r$. The worst-case scenario is that all the clean elements come in the beginning since the algorithm needs to remove more elements without knowing whether it should. After all $\ell_r$ elements have arrived, there are $k - \ell_r$ stale elements that the algorithm unfortunately does not know. The expected number of misses of the algorithm is the expected number of stale elements that are not in the cache when requested. Let’s think of this probability for the $i$-th such element. Every clean element removes another element. If it is stale and before the position $k - i - \ell_r$, another element will be removed when its turn comes. So, for the $i$-th stale element, each clean element corresponds to the removal of one element in the last $k - i + 1$ slots and this is uniformly at random. Since there are $\ell_r$ of those, the probability that $i$-th element will be the one removed is at most $\frac{\ell_r}{k - i + 1}$ for the $i$-th time where $i \in [1, \ell_r - 1]$. Summing over all those positions, the number of cache misses of Marker in this phase is at most $\ell_r H_k$.

**B Proof of Lemma 3**

In this section, we provide the proof of the lemma connecting spread to absolute and squared loss. Before doing so, we provide a useful auxiliary lemma.

**Lemma 4.** For odd $T = 2n + 1$, one pair $(A_T, B_T)$ minimizing either absolute or squared loss subject to the constraints of the spread definition is $A_{2n+1} = (0 \ldots 2n)$ and $B_T = (n \ldots n)$.

**Proof.** First we show that there exists a $B_T$ minimizing the loss with $b_i = b_j$ for all $i,j$. Assume otherwise; then there exist two subsequent $i,j$ with $b_i > b_j$. Since $a_i < a_j + 1$ by the assumption on spread, $\min_{x \in b_i,b_j} \{\ell(a_i, b) + \ell(a_j, b)\} < \ell(a_i, b_i) + \ell(a_j, b_j)$. Applying this recursively, we conclude that such a $B_T$ exists.

Second, we show that there exist an $A_T$ that consists of elements $a_{i+1} = a_i + 1$. Since the elements of $B_T$ are all equal to $b$, the sequence $\sum_{i=0}^{2n} \ell(a_i, b)$ is minimized for both absolute and squared loss when $a_i = b+i-n$.

Last, the exact value of $b$ does not make a difference and therefore we can set it to be $b = n$ concluding the lemma.

**Lemma 3** restated: For absolute loss, $\ell_1(A, B) = \sum_i |a_i - b_i|$, the spread of $\ell_1$ is $S_{\ell_1}(m) \leq \sqrt{4m + 1}$. For squared loss, $\ell_2(A, B) = \sum(a_i - b_i)^2$, the spread of $\ell_2$ is $S_{\ell_2}(m) \leq \sqrt{14m}$.
Proof. It will be easier to restrict ourselves to odd $T = 2n + 1$ and also assume that $T \geq 3$. This will give an upper bound on the spread (which is tight up to small constant factors). By Lemma 4, a pair of sequence minimizing absolute/squared loss is $A_T = (0, \ldots, 2n)$ and $B_T = (n, \ldots, n)$. We now provide bounds on the spread based on this sequence, that is we find a $T = 2n + 1$ that satisfies the inequality $\ell(A_T, B_T) \leq m$.

**Absolute loss:** The absolute loss of the above sequence is:

$$\ell(A_T, B_T) = 2 \cdot \sum_{j=1}^{n} j = 2 \cdot \frac{n(n + 1)}{2} = n(n + 1) = \frac{T - 1}{2} \cdot \frac{T + 1}{2} = \frac{T^2 - 1}{4}.$$ 

A $T$ that makes $\ell(A_T, B_T) \geq m$ is $T = \sqrt{4m + 1}$. Therefore, for absolute loss $S_\ell(m) \leq \sqrt{4m + 1}$.

**Squared loss:** The squared loss of the above sequence is:

$$\ell(A_T, B_T) = 2 \cdot \sum_{j=1}^{n} j^2 = 2 \cdot \frac{n(n + 1)(2n + 1)}{6} = \frac{(T^2 - 1) \cdot T}{12} = \frac{T^3 - T}{12} \geq \frac{8T^3}{9 \cdot 12} = \frac{2T^3}{27}$$

where the inequality holds because $T \geq 3$.

A $T$ that makes $\ell(A_T, B_T) \geq m$ is $T = \sqrt[3]{14m}$. Therefore, for squared loss $S_\ell(m) \leq \sqrt[3]{14m}$. 

\qed