Motivation: Machine Learning can be reformulated as reducing uncertainty: the better your model is, the more certain you are about an outcome. In the context of online algorithms, greater certainty can be directly used to obtain with a lower competitive ratio $c$. As a result, if one is able to reduce uncertainty, then you can directly minimize the lower bound on your competitive ratio.

Therefore, this lecture explores combining learned models with online algorithms, in efforts to reduce the competitive ratio of the online algorithm. We can partition our understanding of online algorithms into two subsections:

1. First, if the ML prediction is good, then you can trust this prediction over the heuristics defined in your algorithm. This is desirable.

2. Secondly, if the ML prediction is bad, then you should revert to the non-augmented algorithm. In this case, you don’t lose anything by having a prediction, so this is also desirable.

This all sounds great, but the question becomes: how do you know if the ML prediction is bad? This is typically a hard thing to classify. We address this in section 3.

1 Example: Caching

Online algorithms have a lot of roots in caching, so let’s begin by examining caching algorithms.

A caching problem usually has the following formulation:

You have a cache of size $k$, and elements arrive one at a time. One of two scenarios may occur:

1. If the arriving element is in the cache, then the cost is zero.

2. If the arriving element is not in the cache, then you incur a cache miss, and pay a cost of one. You must evict one item from the cache, and place the new element in its slot.

One implication of these scenarios that any deterministic algorithm must be $k$-competitive. Consider the scenario where if the algorithm makes a single mistake, then it may evict the item that was going to occur next. This requires emptying the entire cache over $k$ items, and a cost of $k$ will occur over the optimal algorithm.

While randomized algorithms exist that are $O(\log k)$ competitive, no constant competitive ratio exists. Our goal is to find a constant competitive algorithm, and use theory to guide our selection.

However, it is quite difficult to learn an oracle that provides the best predictor to any unstructured problem; instead, one needs to create structure that makes the problem tractable. In order to provide some structure to our problem, let us consider what an optimal offline solution may be characterized by.
2 Offline Solution

In order to determine the optimal offline solution, we resort to Belady’s Rule. If you go ahead and evict the item that appears furthest in the future, then you should evict the fewest number of items as compared to any other algorithm. This makes intuitive sense: you are removing the item from the cache that will be needed the least, and therefore, utilizing each element in the cache to the highest degree.

This motivates a learned approach: can you predict at what time each item will appear? This formulation also doesn’t motivate a need to worry about the consistency of predictions between timesteps, which is quite useful for our application.

3 Determining Prediction Quality

As noted in the problem motivation, a lot of this sounds great in theory. However, how does one actually quantify the prediction quality?

Thankfully, we can align this with our conception of what a good predictor is. In other words, using a smart choice for your loss function may help find a mapping that performs well on ground truth data. For most of this formulation, the setup to train the model is standard.

4 Building the Model

It is quite tempting to use the performance of the predictor, $h$, in the caching algorithm. However, this couples the quality of the predictor with that of the caching algorithm.

The authors advocate for a more practical approach. That is, decompose the problem into two parts:

1. Find a good prediction, which can be delegated towards model training.
2. Use this prediction effectively in an online algorithm, which can be delegated towards algorithm designers.

If we want to evict the item that appears furthest in the future, then it seems reasonable to minimize the difference between when the element appears and when we predict the item will appear. As a result, we consider minimizing the $L_1$ norm between the absolute arrival time and the predicted arrival time.

In other words,

$$\eta = \sum_i |h(i) - t(i)|$$

where $h(i)$ is the predicted arrival time, and $t(i)$ is the actual arrival time.

Alright, now we have an oracle that can provide us with the predictions for which elements occur at which times. How can we use this?
5 Blindly Following the Oracle

Let’s say you blindly follow an oracle, but what if that element never occurs? Let’s examine this scenario below.

\[ c \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ y \ x \ ... \ c \]

In other words, \( x \) appears at position \( 2r \), \( y \) appears at position \( 2r + 1 \), and \( c \) appears at positions 1, \( T \). Let’s say we expected \( c \) to appear at time 1. In this situation, our cache would appear as the following:

- \( t=2 \), Your cache exists of \([c, x]\)
- \( t=3 \), Evict \( x \), add \( y \): \([c, y]\)
- \( t=4 \), Evict \( y \), place \( x \): \([c, y]\)

It is clear that \( c \) should be evicted, but this never happens since we never evicted elements which have already passed. This leads to constant error on average.

What if we evict the element whose predicted arrival time has passed? If no element has already passed, then we predict the item that would occur furthest in the future.

6 Removing Past Predictions

\[ a \ x \ a \ y \ a \ a \ a \ x \ a \ a \ a \ a \ a \ a \ a \ y \ a \ a \ ... \ c \]

Let’s assume that \( x, y \) will always be predicted to arrive at \( T \), and \( c \) will be predicted to arrive at time \( T - 1 \). \( a \) will always be predicted to arrive at the correct time.

The cache in this situation will look like:

- \( t=0 \), \([a, c, y]\)
- \( t=2 \), Evict \( y \), place \( x \): \([a, c, x]\)
- \( t=4 \), Evict \( x \), place \( y \): \([a, c, y]\)

Since \( x \) occurs at positions \( 2^{2r-1} \) and \( y \) occur in positions \( 2^{2r} \), this means that there will be \( O(\log T) \) misses overall.

If we compare this against the optimal solution:

1. \( t=0 \), Initial cache: \([a, c, y]\)
2. \( t=2 \), Evict \( c \), place \( x \): \([a, x, y]\)
3. \( t=T \), Evict \( x \), place \( c \): \([a, c, y]\)
This leads to two misses overall.

On average, prediction error will still be constant; this can lead to a super-constant competitive ratio.

7 Learning-Augmented Algorithm

7.1 Marker Algorithm

The Marker algorithm [Fiat et al.] is used to choose which element to evict from a full cache. There is no learning involved in the Marker algorithm. The final algorithm will augment the Marker algorithm with the model we developed in Section 4. The Marker algorithm leaves all the elements in the full cache unmarked in the beginning. As an element arrives, mark the element. When you need to evict an element, evict a random unmarked element in the cache. If all the elements in the cache are marked, unmark all the elements. This algorithm is $2 \log k$ competitive where $k$ is the size of the cache.

To provide some intuition on how the Marker algorithm’s competitive ratio was calculated, let’s define a few terms. A phase is a period in which a series of elements causes the cache to reset all the marks. We look at a fixed phase $i$ to determine which element are ‘stale’ and which ones are ‘clean’. A stale element is one that was marked in phase $i-1$, and appears again in phase $i$. A clean element is one that is not stale. Since stale elements will not be evicted using the Marker algorithm (since they are marked), we look to the number of evictions of clean elements to get a lower bound on the optimal algorithm. For each phase, if a clean element is not in the cache, it will occur a miss in this phase. If it is in the cache, then it must have had a miss in the previous phase. Let $c$ be the total number of clean elements. There for $c$ clean element in each phase $i$, there are at least $\frac{c}{2}$ cache misses per phase in the optimal scenario. If we sum this over all phases:

$$\sum_i \frac{c_i}{2} = \frac{c}{2}$$

So we will get at least $\frac{c}{2}$ in the optimal offline algorithm. For the Marker algorithm, there are two types of ways to get a cache miss: 1) a request for a clean element, and 2) a request for a stale element that has been evicted in the current phase. At most, the probability for having evicted the $i^{th}$ stale element request is $\frac{c}{k+1-i}$. Summing over all possible stale element requests, we get the expected number of cache misses from stale requests to be approximately $c \log k$. Adding in the number of cache misses from clean requests gives us a total of $c \log k + c$. By analyzing the number of cache misses between the Marker algorithm and the optimal offline algorithm, we get a competitive ratio of $2 \log k$.

7.2 Predictive Marker Algorithm

The predictive Marker algorithm augments the Marker algorithm by deciding to use the model in Section 4 instead of evicting a random unmarked element when there is a cache miss. When we see and element that is not in the cache, we look at all the unmarked elements in the cache, and then evict the element in which model outputs to latest predicted arrival time. If the predictions of the model are close to perfect, we will get an algorithm close to competitive ratio of Belady’s Rule.
Our perfect model will give our cache the property that stale elements will never cause misses, but clean elements will always cause misses since they were not present at the beginning of the phase. Therefore, the optimal algorithm will get, on average, half the misses for each clean element based on the reasoning in 7.1. This means that if the predictions of the model are close to perfect, we will get a 2-competitive algorithm. However, if the model’s predictions are extremely inaccurate, our augmented algorithm will be $k$-competitive.

8 Analysis of Predictive Marker Algorithm

Let’s create a graph of elements that resulted in a cache miss. Suppose element $a$ incurred a cache miss. Either $a$ was a clean element, or $a$ was a stale element that was previously evicted by element $b$. In our graph, if $a$ was a stale element, we would add an edge from element $a$ to element $b$.

Each edge corresponds to the element that causes its eviction. We can call this graph a blame graph. The graph is a series of chains. As we follow the directed edges, we can see the roots at the end of the chains. Each chain ends in a clean element, so the total number of clean elements equals the number of chains in the graph. The total length of the chains is the total number of cache misses incurred. Since every element in the chain causes a cache miss and the root is the only clean element in its chain, the root has to be the first element in its chain to arrive in phase $i$.

The root of the chain is a clean element and the rest of the nodes are stale elements. In Figure 3, $d$ is the clean element and $a$, $b$, and $c$ are the stale elements. Looking at the chain in Figure 3, our algorithm chose to evict $c$ when $d$ arrived and chose to evict $b$ when $c$ arrived; so on and so forth.
Therefore, our model predicted the order of elements to arrive to be \( a, b, \) then \( c, \) but the actual arrivals came as \( c, b, \) then \( a. \) Therefore, on a chain of length \( t, \) the calculated error is \( \Omega(t^2). \) If our model had a prediction error of \( \eta, \) then the length of any chain can be at most \( O(\sqrt{\eta}). \) If we split the error across each chain equally, the competitive ratio becomes \( O(\sqrt{\eta/c}). \)

We make another addition to the predictive Marker algorithm just in case the model is inaccurate. If the length of the chain grows past what the competitive ratio of the regular Marker algorithm would be, then the algorithm will switch to the regular Marker algorithm and discard the model. This gives a final competitive ratio for our algorithm:

\[
\min \left( 2 + 2\sqrt{1 + \frac{4\eta}{c}}, 4\log k \right)
\]

When our error (\( \eta \)) is large, we fall back to the randomized strategy and get a \( \log k \) competitive strategy. When our error is small, we get a constant competitive algorithm.

### 9 Experimental Results

LRU (Least Recently Used) is a popular caching strategy. This evicts the element in the cache that was least recently used. In fact, LRU is a predictive marker strategy with a different kind of model. We can think of LRU’s predictive model as a function that takes an input element that came at time \( i \) and returns a prediction of \(-i\). To see how our predictive Marker algorithm compares to LRU, we can look to Brightkite data and Citibike data. Brightkite data is location data for a social media app and the Citibike data is the timestamped bike docks and undocks. Lykouris and Vassilvitskii [Lykouris et al.] look at the real data and then add synthetic log-normally distributed noise to the data. They add noise to the data to make see how noise affects the quality of the predictor and log-normally distributed noise will let us see how the predictor handles very large errors. The following table shows the empirical results:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Britekite Competitive ratio</th>
<th>Citi Bike Competitive ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>BlindOracle</td>
<td>2.049</td>
<td>2.023</td>
</tr>
<tr>
<td>LRU</td>
<td>1.280</td>
<td>1.859</td>
</tr>
<tr>
<td>Marker</td>
<td>1.310</td>
<td>1.869</td>
</tr>
<tr>
<td>Predictive Marker</td>
<td>1.266</td>
<td>1.810</td>
</tr>
</tbody>
</table>

Figure 4: The empirical results from the Brightkite and Citibike data.
We can see that the predictive Marker strategy reduces the most number of cache misses and performs better than LRU. As we recognized earlier, the blind oracle relies heavily on the model’s predictive ability. The following chart shows how these different strategies perform against the synthetic data:

Figure 5: The empirical results from the Brightkite and Citibike data.

Even as we increase the error parameter in the data, the predictive Marker algorithm still outperforms LRU. As we can see, the blind oracle is very sensitive to errors in the data.

10 Further Research

Further research would be to repeat the experiment above with a more extensive predictive model. Only 4 parameters were used and there was no hyper parameter tuning of the model. It would also be useful to try the same experiment on different data sets to see if the results hold. Furthermore, an extensive analysis of other learning-augment online algorithms should be considered as well.

References
