Online Caching through Optimistic Online Mirror Descent

Gijs Admiraal
Supervisor(s): George Iosifidis, Naram Mhaisen EEMCS,
Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The advent of wireless networks such as content distribution networks and edge computing networks calls for more effective online caching policies. Traditional policies lose performance since these new networks deal with highly non-stationary requests and frequent popularity shifts. Consequently, a new framework called Online Convex Optimization (OCO), which does not assume the request pattern, has recently been used to tackle the online caching problem. Besides, in many practical scenarios, a request prediction of unknown quality is available. This paper will leverage that and proposes a new online caching policy that uses these predictions. This policy will use the Optimistic Online Mirror Descent (OOMD) algorithm to solve the OCO problem. The policy will still obtain the same regret bound as its non-optimistic counterpart up to some constant even if the predictions are not accurate. The performance of the proposed policy is evaluated and compared with previous OCO-based policies with the use of trace-driven numerical tests.

1 Introduction

1.1 Background

The problem of caching is concerned with utilizing fast but limited storage in the best possible way. Although caching was first developed in the 1960s [4] [17] for local computer systems, its use has been widely adopted after the explosion of web traffic [5] and the arrival of Content Distribution Networks (CDN) and cloud services.

Since serving requests (i.e., fetching files) from the cache is faster and more efficient, a caching optimization system or caching policy aims to select a subset of files to be placed in the cache to maximize the number of requests that can be served from that cache. This reduces the load and latency on and between servers and consequently increases the overall performance of a system.

Different caching policies have been proposed throughout the years [21], each trying to ensure that the necessary files are in the cache storage when requested. An example of such a policy is the Least-Recently-Used (LRU) policy which evicts the file from the cache which has least recently been used and inserts the newly requested file. Another such policy is the Least-Frequently-Used (LFU) policy which evicts and places files not based on recency but frequency.

For different types of request patterns, different policies excel [12]. A request pattern can be stationary, where it is assumed that requests are generated as i.i.d samples from a probability distribution with fixed parameters. Request patterns can often in a realistic setting vary over time and be non-stationary. The widely adopted policies such as LRU and LFU lose performance under these time-varying traces [8] [13] [14]. Knowing the optimal caching policy to use in each specific situation requires knowing all future requests. Unfortunately, in a practical setting, this is unknown. It is thus more practical to have a policy which optimizes the hit ratio of the system under any request trace.

An example of caches that receive time-varying and thus highly non-stationary request traces are the emerging edge caches [6] [16] [25]. These caches were designed with the advent of wireless networks such as CDNs and cloud services [7] and are located at the edge of the network. They will relocate the services provided from centralized cloud services to be closer to the users. This complex and dynamic computing architecture is therefore used to improve the responsiveness and reduce the backhaul in traffic for cloud services. A model of such an architecture can be seen in figure 1.

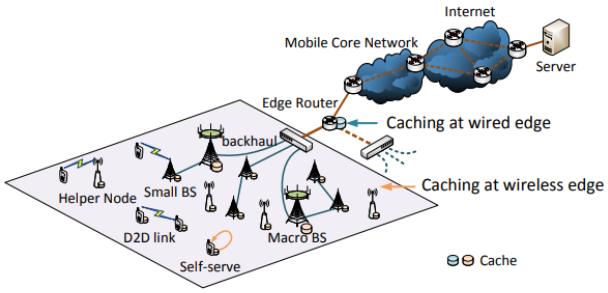


Figure 1: Example model of an edge caching network [1]

Caching policies that adapt under different and non-stationary request patterns have consequently recently been studied. In these studies, the caching problem is approached as an online learning problem [9] [13]. With online learning, a policy learns and alters its cache state after each request to reach the optimal state as time progresses. An online learning policy has no assumption of the request trace. This is opposed to offline learning, which assumes that the request pattern has an unknown distribution and tries to learn this distribution before any request is revealed through pretraining. For the caching problem, it is hence more favourable to use a form of online learning since it places no statistical assumption on the file request pattern and can thus handle non-stationary request distributions. Furthermore, it does not require any pretraining or training data and allows the algorithms to be more scalable.

1.2 Related work

More recently, research has used the mathematical framework of (OCO) [11] to model the online caching problem [20]. The theory of OCO is based on a learning algorithm that guesses at each time slot t a decision vector $\mathbf{x}_t$ from a convex set $\mathcal{X}$ without knowing the performance of this vector. This performance of $\mathbf{x}_t$ is given by the time-slotted performance or cost function $f_t(\mathbf{x})$ which is only revealed after selecting $\mathbf{x}$. The algorithm's goal is to minimize the rate of regret

[1] Modified Figure 1 from D. Liu, B. Chen, C. Yang and A. F. Molisch, "Caching at the wireless edge: design aspects, challenges, and future directions," in IEEE Communications Magazine, vol. 54, no. 9, pp. 22-28, September 2016, doi: 10.1109/MCOM.2016.7565183.

$R_T = \sum_{t=1}^{T} f_t(\mathbf{x}^*) - \sum_{t=1}^{T} f_t(\mathbf{x}_t)$ up until time T. This is calculated by the difference in performance between the chosen $\mathbf{x}$ and the optimal solution $\mathbf{x}^*$. The optimal solution is the solution that would be chosen if the algorithm knows the future and is defined as $\mathbf{x}^* = \arg\min_{\mathbf{x}\in\mathcal{X}} \sum_{t=1}^{T} f_t(\mathbf{x})$. This is hence a suitable framework for the caching problem, where the learning algorithm is the policy and the decision vector is the cache state. The cost function at time t is based on the request received at time t and is thus unknown to the policy before selecting a cache state for time t. An advantage of the OCO framework is that the learning algorithm performance is measured by any arbitrary performance function. The algorithm optimizes any cost function, this function could focus on measuring storage efficiency and the ratio of requested files that the cache can serve from its memory. It could also model the energy efficiency of moving files around in the cache network. This shows therefore the versatility of how the OCO framework can be used to model the online caching problem.

An online caching algorithm that used online gradient descent (OGD) was proposed and proven that it has universally optimal performance and attains a sub-linear regret with time $\mathcal{O}(\sqrt{T})$ [20]. This is called a no-regret algorithm, as their time-average regret becomes negligible when $R_T/T = 0$ as $T \to \infty$.

One later study showed that the generalization of the OGD algorithm, called the online mirror descent (OMD) algorithm could also maintain a sub-linear regret [24]. This OMD policy would still obtain a sub-linear regret when the requests would come in batches. The OMD algorithm can be used to come up with different OCO algorithms through different use of mirror map functions[2].

A new aspect that could improve the performance of these learning-based cache policies is predictions about the forthcoming request. For example, online content platforms such as YouTube and Netflix [10] give users content recommendations. These recommendations could be leveraged in that they can serve as predictions about future requests. This information about future requests could, if properly applied, significantly improve the optimally of the caching policy.

Optimistic OCO algorithms are algorithms that use these predictions. These predictions are used to alter the decision vector $\mathbf{x}_t$ to decrease the cost before the performance function is revealed. The algorithm is said to be "optimistic" about the revealed prediction, considering it as if it were true. A study has already shown that these optimistic algorithms have a regret bound that is the same as the non-optimistic version scaled to some constant depending on the quality of the predictions [22]. When all predictions are correct the average regret drops to a constant. This research gave only generic regret bounds but no algorithm or implementation was proposed as of yet[3].

[2]OMD makes use of an extra *dual space* in which the gradient exists. The mirror map function links this extra space to the *primal space*, in which the variables live.

[3]This research has also not given any optimal learning rate(s) that could satisfy regret bounds. The learning rate(s) plays an essential role in the devising and implementation of actual algorithms.

The most recent research on using optimism in the framework of OCO is an optimistic version of the follow the regularized leader (FTRL) algorithm [19]. It showed that even with faulty recommendations the policy could still maintain a sub-linear regret. However, this study focused on a different algorithmic framework than the one that will be considered in this paper[4].

This paper aims to propose a design of an online caching policy which will be using the optimistic online mirror descent (OOMD) algorithm

The performance of the policy will be benchmarked for different qualities of recommendations and compared to previous OCO policies, OGD and OMD$_{ne}$. The policy will be tested to see if its performance improves as the quality of predictions is higher. Furthermore, the policy should also provide performance bounds depending on the quality of the predictions. The performance of the policies will be characterized by their average cost and regret over various request models.

2 System description

Cache Scenario

The caching scenario that is considered is where a single user makes file requests to a single cache. In this network, a user can request any file from the set $\mathcal{N}$ with size N to the cache. When a cache miss happens for a request then it has to be retrieved from a remote server. The cost of requesting the files is denoted by the vector $\mathbf{w}$ where $w_i \in \mathbb{R}$ for file $i \in \mathcal{N}$. The cost of each file can vary since it may be stored on different remote servers.

Requests

The system operates in discrete time slots, $t = 1, 2, ...T$. Each request contains a single file after which the cache state is updated. A request is represented by the vector $\mathbf{r}_t$ of length N where $r_{t,n} = 1$ when file $n \in \mathcal{N}$ is requested and $r_{t,m} = 0$ where $m \neq n$, $m \in \mathcal{N}$. The request pattern is not known to the system. It can follow different types of request patterns, both fixed and time-varying and can even be an adversary trace that tries to strategically break down the caching operation. The set of all possible request can be described by $\mathcal{R}$ and can be formally be described as $\mathcal{R} = \{\mathbf{r} \in [0,1]^N : \sum_{i=1}^{N} r_i = 1\}$.

Caching

The cache can store at most a total of k files. The cache state is represented by the vector $\mathbf{x}_t$ of length N belonging to the set $\mathcal{X} = \{\mathbf{x} \in [0,1]^N | \sum_{i=1}^{N} x_{t,n} \leq k\}$ for a time slot t. Here $x_{t,n}$ constitutes the proportion of file n stored in the cache state at time t and is capped by the capacity constraint k.

Recommendations

A recommender system gives at the beginning of every time slot t a file request recommendation to the cache. The recommendation is given the same way as a request by the vector $\bar{\mathbf{r}}_t$

[4]A survey has given a detailed discussion and comparison of the advantages and disadvantages of different families of adaptive learning methods [18]

and also exists in the set $\mathcal{R}$. It is assumed that $\bar{\mathbf{r}}_t$ is revealed at the end of time slot $t-1$. The quality of the recommendation or percentage of the time that the recommendation is correct is given by $\sigma \in [0,1]$. Where $\sigma = 1.0$ always gives a perfect recommendation and $\sigma = 0.0$ always gives a faulty recommendation.

Cost Objective Function

When a request $\mathbf{r}_t$ arrives at the cache, then the cache measures its current performance of this request through the following function:

$$f_{\mathbf{r}_t}(\mathbf{x}_t) = \sum_{i=1}^{N} w_i r_{t,i}(1 - x_{t,i}) \tag{1}$$

This formula thus incurs a cost for the fraction of the requested file, $r_{t,i}$, missing from its cache state, $(1-x_{t,i})$. This cost is proportional to the file cost w_i. This objective function (1) can be motivated by several real-life use-cases. Firstly this equation captures the delay that is incurred by the cost of retrieving a file that harms the users' experience. Consequently, this does not only model the users' cost but also the cost on the server or the entire network. Furthermore, this function handles the adaption of the system when it is decided to aggregate requests into batches (e.g. where a file might be requested multiple times or multiple files are requested at once). This is useful since requests are not always handled file by file in real-life applications.

Regret

The caching policy that knows all future requests is called the optimal caching policy and is used to benchmark an online learning policy. Since the optimal policy which has a dynamic cache state has too high of a performance, an optimal policy is used with a static cache state. This cache state $\mathbf{x}^*$ is defined by:

$$\mathbf{x}^* = \text{ arg min}_{\mathbf{x}\in\mathcal{X}} \sum_{t=1}^{T} f_{\mathbf{r}_t}(\mathbf{x}) \tag{2}$$

To benchmark an online learning policy π with other policies the regret of each policy needs to be calculated. This is measured by taking the difference between the static optimal policy and the policy π. This is given in the equation below:

$$R_T(\pi) = \sup_{\{f_t\}_{t=1}^{T}} \left[\sum_{t=1}^{T} f_{\mathbf{r}_t}(\mathbf{x}_t) - \sum_{t=1}^{T} f_{\mathbf{r}_t}(\mathbf{x}^*) \right] \tag{3}$$

3 OCO based Policies

First the existing OGD and OMD_{ne} caching policies are highlighted. Then the implementation of the optimistic online caching policy is given.

Notation	Description
$\mathcal{N}$	Set of the file catalogue
N	Size of the set $\mathcal{N}$
$\mathcal{R}$	Set of all possible requests
$\mathbf{r}_t$	Vector representing request at time t
$\bar{\mathbf{r}}_t$	Vector representing recommendation at time t
σ	Quality of recommendation
T	Time horizon
$\mathcal{X}$	Set of all possible cache states
$\mathbf{x}_t$	Vector representing cache state at time t
$\mathbf{x}^*$	Vector representing the optimal static cache state
k	Cache capacity
η	Learning rate
$\mathbf{w}$	Vector representing file cost
$f_{\mathbf{r}_t}$	Cost function for request $\mathbf{r}_t$
π	Online cache policy
$R_T(\pi)$	Regret of the cache policy π

Table 1: Table containing all important notation

Online Gradient Descent (OGD)

The OGD algorithm works as a caching policy as follows. The cache is initialized with a feasible state $\mathbf{x}_1 \in \mathcal{X}$ and is updated after every request $\mathbf{r}_t$. A cost is derived from the current cache state $f_{\mathbf{r}_t}(\mathbf{x}_t)$ and the cache state is updated to $\mathbf{x}_{t+1}$ as such:

$$\mathbf{x}_{t+1} = \Pi_{\mathcal{X}}(\mathbf{x}_t - \eta \nabla f_{\mathbf{r}_t}(\mathbf{x}_t)), \quad \text{for all } \ t \in [T-1] \tag{4}$$

Where $\Pi_{\mathcal{X}}(\cdot)$ is a Euclidean projection back onto $\mathcal{X}$ since after an update the cache state might exceed the capacity constraint. The learning rate is $\eta \in \mathbb{R}$ which scales the step-size of the OGD algorithm. Recent studies have shown that OGD is an effective caching policy [20].

Online Mirror Descent$_{\text{ne}}$ (OMD_{ne})

The Online Mirror Descent (OMD) algorithm [11] is the online version of the mirror descent (MD) algorithm [3]. Furthermore, OMD is the abstract version, or generalization, of the OGD algorithm. The difference between the two is that the cache state lives in the primal space and the gradients in the dual space. The spaces are connected by a differentiable mirror map $\Phi(\mathbf{x})$, a function that inverts the update from the dual space onto a change on the primal space. This mirror map function allows the OMD algorithm to recover to its derivations such as the OGD algorithm.

A current study has also already shown the effectiveness of the OMD algorithm as a caching policy [24]. This research showed that when OMD is used with the negative entropy mirror map, $\Phi(\mathbf{x}) = \sum_{i=1}^{N} x_i \text{log}(x_i)$, the algorithm has only two steps and will be denoted as OMD_{ne}. Where the first step does an update on the current cache state using the cost function $f_{\mathbf{r}_t}(\mathbf{x}_t)$ and a set learning rate η. This first step is shown in 5 where it can be seen that this cache state updates via a multiplicative rule as opposed to an additive rule for OGD.

$$\hat{x}_{t+1,i} = x_{t,i} e^{-\eta \frac{\partial f_{\mathbf{r}_t}(\mathbf{x}_t)}{\partial x_i}} \tag{5}$$

The second step, is projecting this updated cache state y_{t+1} back by using a Bregman divergence [15] associated with the negative entropy mirror map and is given by $\Pi_{\mathcal{X}}^{\Phi}(\cdot)$. This is different to the orthogonal projection used in OGD. The full OMD_{ne} cache update looks as such:

$$\mathbf{x}_{t+1} = \Pi_{\mathcal{X}}^{\Phi}(\mathbf{x}_t e^{\eta \nabla f_{\mathbf{r}_t}(\mathbf{x}_t)}), \quad \text{for all} \quad t \in [T-1] \tag{6}$$

Optimistic OMD_{ne} (OOMD_{ne})

The implementation of the OOMD_{ne} algorithm [22] policy proceeds in two steps, using the negative entropy mirror map. One step for the revealed request $\mathbf{r}_t$ and one for the given recommendation $\bar{\mathbf{r}}_{t+1}$. The first step is similar to the OMD_{ne} policy cache state update. The actual request is given to the policy and a proxy cache state is updated to $\mathbf{y}_{t+1}$ using the multiplicative rule and the negative entropy Bregman projection. This first update is scaled by the static learning rate η_1. This step is given below:

$$\mathbf{y}_{t+1} = \Pi_{\mathcal{X}}^{\Phi}(\mathbf{y}_t e^{\eta_1 \nabla f_{\mathbf{r}_t}(\mathbf{x}_t)}) \tag{7}$$

In the second step, a recommendation of unknown quality σ is revealed and this leads to another multiplicative update on the new proxy cache state $\mathbf{y}_{t+1}$. This new cache state $\hat{\mathbf{x}}_{t+1}$ is then projected back to the final cache state $\mathbf{x}_{t+1}$. This update is scaled by the learning rate η_2 which is recalculated at every time step t by the algorithm. The policy tracks the ratio of correct recommendations, $\hat{\sigma}$, given up to time t. This estimation is then used to find η_2 by the following equation[5]:

$$\eta_2 = \frac{\eta_1}{(1-\hat{\sigma})} \tag{8}$$

The learning rate, η_2, is increased based on the quality of the recommendations. The higher the quality of recommendations, the higher the learning rate would be. If the algorithm can trust the predictions it can thus more confidently update the cache state using the larger learning rate.

Convergence to a faulty cache state by wrong predictions is mitigated since the first update always occurs on the proxy cache state and not on the actual cache state. A previous wrong prediction at time $t-1$ will thus only affect the cache state at time t and not at time $t+1$

The full algorithm for OOMD_{ne} as a caching policy is shown in algorithm 1 and a visual representation is shown in figure 2

Algorithm 1 Optimistic Online Mirror $\text{Descent}_{\text{ne}}$

Require: $\mathbf{x}_1 = \arg\min_{\mathbf{x}\in\mathcal{X}} \Phi(\mathbf{x}), \eta_1 \in \mathbb{R}_+$

for $t \leftarrow 1, 2, \ldots T$ **do**
- $\hat{\mathbf{y}}_{t+1} \leftarrow \mathbf{y}_t \cdot e^{\eta_1 \mathbf{r}_t \mathbf{w}}$ ▷ Update proxy state based on the request
- $\mathbf{y}_{t+1} \leftarrow \Pi_{\mathcal{X}}^{\Phi}(\hat{\mathbf{y}}_{t+1})$ ▷ Project proxy state back to feasible region $\mathcal{X}$
- $\hat{\mathbf{x}}_{t+1} \leftarrow \mathbf{y}_{t+1} \cdot e^{\eta_2 \bar{\mathbf{r}}_t \mathbf{w}}$ ▷ Update cache state based on recom. and proxy state
- $\mathbf{x}_{t+1} \leftarrow \Pi_{\mathcal{X}}^{\Phi}(\hat{\mathbf{x}}_{t+1})$ ▷ Project cache state back to feasible region $\mathcal{X}$

end for

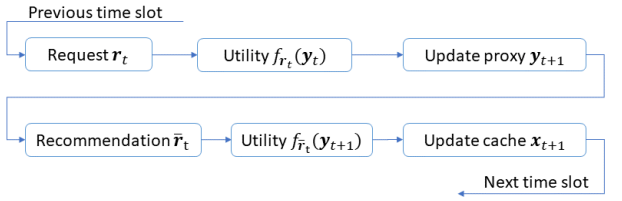


Figure 2: Visual representation of how at each time slot the request and recommendation is processed

4 Results

4.1 Experimental Setup

A modular caching simulator is built in Python 3. This simulator can receive two different types of input. The first is the option which indicates which policies should be tested in the simulator, OGD, OMD_{ne} and OOMD_{ne}. Each policy is implemented as a separate class and each is extended from an abstract cache class. The second input can receive either a synthetic request trace that is either randomly generated or a real trace taken from a dataset. These synthetic traces can mimic real-life traces such as shifting popularity traces, Poisson-shot model traces and adversarial traces amongst other types of distributions [23]. The real trace is taken from a MovieLens dataset [1]. Each trace is defined as a $N \times T$ matrix where each row represents a time slot where the requested file is one-hot encoded. As an output, the simulator gives the results of each policy in the given scenario. These results on which the policies are benchmarked are the cost acquired for each time step and its regret.

Policies

The policies that are benchmarked against each other are the OGD policy, the OMD_{ne}, two different OOMD_{ne} policies, and the static Optimal policy. The OGD and OMD_{ne} policies are used as a base to compare the OOMD_{ne} policies. The difference between the OOMD_{ne} policies is the rate at which correct recommendations are given. One OOMD_{ne} policy has a rate of 20% correct predictions. This is to show the performance of the policy in a situation with mostly bad predictions. Contrary to a mostly faulty recommendations policy is the OOMD_{ne} with 80% correct predictions, which shows the performance of the policy in a more ideal scenario. The Optimal Policy is the static cache state that is selected to minimize the total cost over the whole request trace.

Traces

The policies will be tested over stationary request traces, non-stationary request traces, adversarial traces and a MovieLens trace. Each entry w_i will be equal to 1 for the file cost vector $\mathbf{w}$ on each experiment.

There will be two experiments done on the stationary request traces. The first will be according to a *Zipf* distribution with exponents $\alpha = 0.6$ and has a cache configuration that has a catalogue size of $N = 1000$, a cache size of $k = 100$ and a time slot size of $T = 5000$. For this experiment, policies with different learning rates will be compared.

[5] In the experimental setup a small constant is added to prevent dividing by zero

The second experiment will be according to different stationary requests that follow a Zipf distribution with exponents $\alpha \in \{0.2, 0.7, 1.2\}$ but will maintain the same cache configuration. The exponent α will manage the diversity of file requests over the trace. An $\alpha = 0$ will correspond to each file being requested with an equal chance. The higher the exponent value becomes the higher the probability will be for certain file requests, leading to more popularity in a trace.

One experiment will be done for a non-stationary request trace which will have a catalog of $N = 2000$ from which request will be sampled according to a Zipf distribution with exponent $a = 0.8$. At every 1500 request will a shift in file popularity occur where file $i \in \{1, ...N\}$ assumes the popularity of file $j = (1 + (i + N/5) \bmod N)$. The total time window would thus be $T = 7500$ and the cache will be set to $k = 150$.

The adversarial trace is a trace that is sampled to work against the cache state and has no set popularity making it a highly non-stationary trace. The trace *slides over the catalogue*, starting with a random file $i \in \{1, ...N\}$. Then at the next time slot, the next file $i + 1$ is requested. This continues for every time slot until file N is reached, after which this pattern recommences from file 1 until the end of the trace. The cache configuration has a catalog size of $N = 1000$, a cache size of $k = 100$ and a time slot size of $T = 5000$.

Lastly, the performance will be measured on a real-life MovieLens trace. It is obtained from a dataset which includes time-stamped movie reviews. These reviews follow certain different movie popularity's and can hence be used as a realistic non-stationary request trace.

Performance metrics

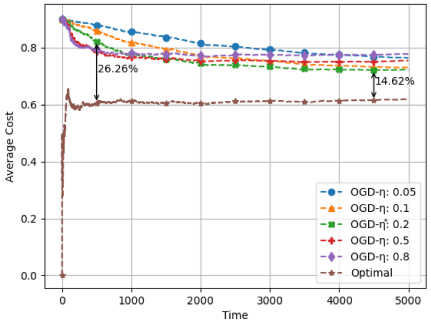
To measure the performance of each policy three different metrics are used. The first is the average cost at time t which is calculated as $\frac{1}{t}\sum_{s=1}^{t} f_{\mathbf{r}_s}(\mathbf{x}_s)$. Another performance metric is the moving average cost and is obtained as follows: $\frac{1}{\min(\tau,t)}\sum_{s=t-\min(\tau,t)}^{t} f_{\mathbf{r}_s}(\mathbf{x}_s)$. In these experiments is $\tau = 500$. Lastly, the performance of a policy π is also measured in terms of the average regret, $\frac{1}{t}R_t(\pi)$. For each of these metrics holds that the lower the score the better.

4.2 Performance

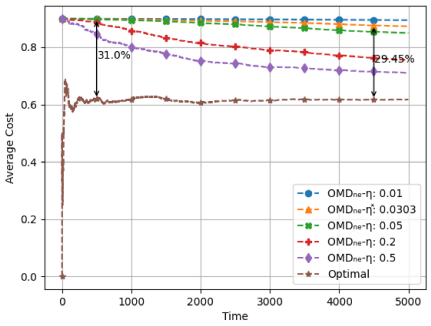
Stationary request traces

In figure 3 (a)-(c) can the effect of different learning rates be seen on the performance of the online policies over stationary request traces. In each figure is the optimal learning rate denoted by η^*. This learning rate is set so that over *any* request trace the performance of the policy will still obtain a sub-linear regret. When an online learning policy has a higher learning rate then it assumes that the past requests are a good indication for the future and will eagerly converge to a cache state that suits these past requests. Consequently, a higher learning rate works well for stationary requests. A lower learning rate is more distrusting and will move slowly towards a cache state that suits the past requests, this is hence better for more non-stationary requests.

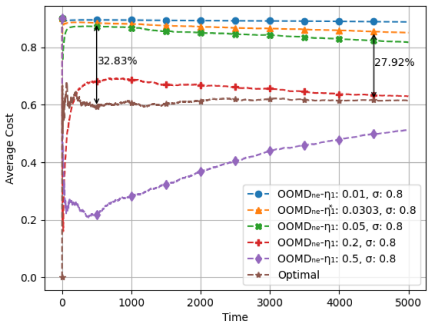
In sub-figure (a) is the optimal learning rate on this cache configuration for OGD $\eta^* = ||\mathbf{w}||_\infty\sqrt{2k/T} = 0.2$ [20]. It can be seen that this learning rate even works optimally over a stationary request. Of all the learning rates does it converge to the lowest average cost at the end of the trace. Additionally, it can be seen from the annotations that as T gets larger the difference for average cost compared to the optimal policy becomes lower. The percentage difference compared to the optimal policy moves from 26.26% to 14.62%, thus showing that over time it can still attain a sub-linear regret.



(a) Average Cost of OGD



(b) Average Cost of OMD

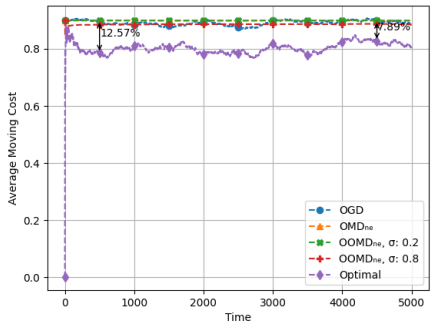


(c) Average Cost of OOMD

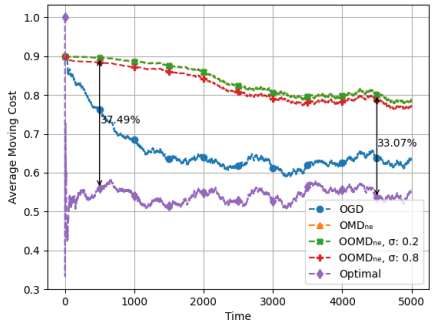
Figure 3: Average cost of the different policies for different learning rates over a *stationary* trace. Each figure shows that a lower learning rate converges slower while a large learning rate converges faster. A fast convergence is not always ideal as seen in sub-figure (a) where a too high of a learning rate does not always reach the lowest average cost. In each sub-figure is the optimal learning rate denoted as η^*

Sub-figure (b) and (c) shows the average cost of the OMD_{ne} policy and the OOMD_{ne} with $\sigma = 0.8$ policy using different learning rates. The optimal learning rate for both policies is $\eta^* = \sqrt{\frac{2*\log(N/k)}{||\mathbf{w}||_\infty T}} = 0.0303$ [24]. Even though the optimal

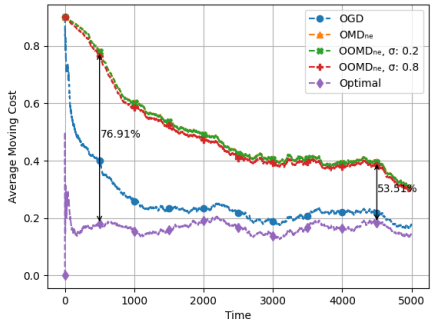
learning rates do not obtain the lowest average cost at the end of the trace, they still have an average cost that is decreasing with time. In sub-figure (b) has OMD_{ne} a small cost reduction from 31.00% to 29.45% and in sub-figure (c) has OOMD_{ne} with $\sigma = 0.8$ a reduction from 32.83% to 27.92%. What can be concluded from sub-figure (c) is that for this trace a learning rate of 0.2 quickly converges to an average cost equivalent to the optimal cache policy. Furthermore, a learning rate of 0.5 has an average cost which is lower than the optimal policy over the whole trace and thus has a negative regret over this trace.



(a) Moving Average Cost with $\alpha = 0.2$



(b) Moving Average Cost with $\alpha = 0.7$



(c) Moving Average Cost with $\alpha = 1.2$

Figure 4: The moving average cost of the policies over traces that follow a *Zipf* distributions with exponents $\alpha \in \{0.2, 0.7, 1.2\}$. The higher the value of the exponent α the lower the diversity of request. It can be seen that OGD obtains a lower moving average cost over the lower diversity traces than OMD_{ne} and OOMD_{ne} with $\sigma = 0.2$ and 0.8

In figure 4 is the performance given over stationary request traces with different levels of diversity. Sub-figure (a) shows that over a trace with low diversity that all online learning policies but the optimal policy have a similar performance. At certain time intervals does OOMD_{ne} with $\sigma = 0.8$ outperform the other policies and at other intervals does OGD have the better performance. A slow convergence for all policies towards the optimal cache state can be seen over the duration of the whole trace. The percentage difference in moving average cost between OOMD_{ne} with $\sigma = 0.2$ and the optimal lowers from 12.57% to 7.89%.

For the sub-figure with a lower diversity (b) and (c) can it be seen that the policies converge to an optimal cache state quicker. The percentages that are shown in these sub-figures show again the percentage difference between OOMD_{ne} with $\sigma = 0.2$ and the optimal policy. It can be found that the lower the diversity the greater the decrease in percentage difference. Furthermore, what is notable is that for these sub-figures the OOMD_{ne} with $\sigma = 0.2$ and OMD_{ne} have a similar performance. This is expected since the predicted file requests only update the proxy state and not the cache state allowing the policy to recover from bad recommendations. Their equal performance shows thus that even with mostly faulty predictions an optimistic policy still has a similar regret bound as its non-optimistic counterpart. When the optimistic policy receives mostly correct predictions, OOMD_{ne} with $\sigma = 0.8$, then it converges faster than OMD_{ne} towards the optimal cache state. What is unexpected is that a greater increase in performance was expected with a higher quality of recommendations. This loss in performance can be because the second learning rate η_2 is not increased enough to make a large difference.

The sub-figures (b) and (c) also show that the OGD policy has a very strong performance compared to the other learning policies as it has lower average moving costs. From sub-figure (c) can it be concluded that a lower diversity in file requests makes OGD quickly converge to the static optimal cache state.

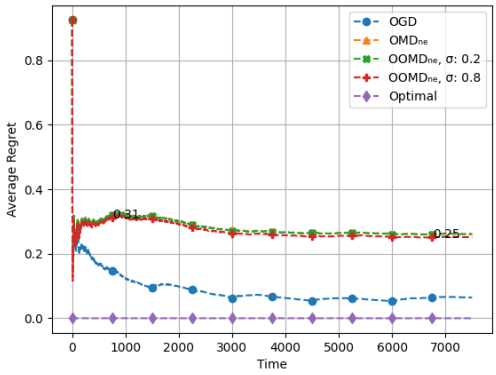
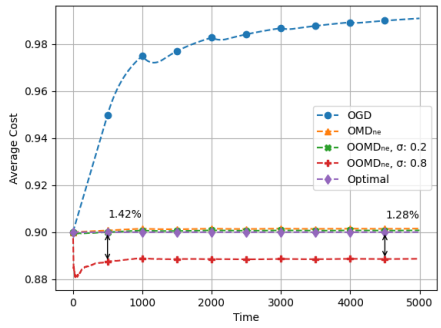


Figure 5: The average regret over a *non-stationary* request trace.

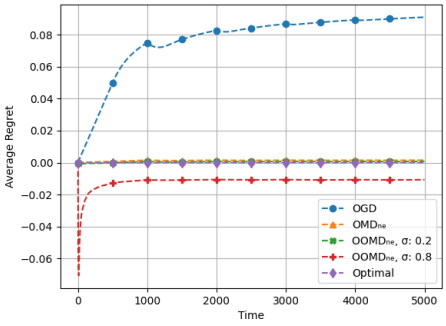
Non-stationary request trace

The figure 5 shows the performance over a non-stationary trace. It is again noticeable that the OMD_{ne} policy and the OMD_{ne} policies have again a similar performance. At two

time-slots has the average regret for OOMD_{ne} with $\sigma = 0.8$ been highlighted. It can be seen that its average regret is decreasing over time from 0.31 to 0.25. The OGD policy performs quite well and gets already close to zero regret at the end of the trace. The increase in the average regret for online learning policies can be seen in the first shift in popularity. At $T = 1500$ the average regret increases slightly but eventually falls again. The other shifts have decreasingly less impact on the average regret as t increases.



(a) The average cost over an adversarial request trace



(b) The average regret over an adversarial request trace

Figure 6: The performance of the online policies over an *adversarial* request trace. The optimistic policy with a high quality of recommendations performs better than the optimal cache state while the OGD policy has very poor performance

Adversarial trace

The strength of the OOMD_{ne} can be seen in Figure 6. The optimistic policy with 80% correct predictions has a lower average cost than optimal as seen in sub-figure (a) and hence has a negative regret as seen in sub-figure (b). This is not surprising because for this trace the optimal static cache state has and the starting cache state has an even distribution over all files, every value of x_i has the same value of k/N. For each recommendation of file i, the cache state slightly increases the proportion of file i. Thus if OOMD_{ne} receives a correct prediction for file i then its proportion of that file is larger than k/N. This will hence result in a lower cost than the optimal. Then because for the next time step the proxy state and not the cache state is used will this correction not affect the next request. It can be seen from both sub-figures 6 (a) and (b) that a lower quality of recommendations is not enough to obtain a negative regret for this trace. Both the OMD_{ne} and the OOMD_{ne} with $\sigma = 0.2$ have a similar performance as the optimal static cache policy.

What is unforeseen from figure 6 is that the optimal learning rate for the OGD policy is too high since its average cost and regret seem to first increase and then plateau.

MovieLens trace

The performance over the non-stationary MovieLens trace in figure 7 eyes similar as to the figure in 5. It can be seen that for this MovieLens trace the optimistic and non-optimistic OMD_{ne} perform alike. Their average regret seems to slowly decrease to the optimal level. The OGD policy again has a better performance in terms of average regret. It quickly converges down before $T = 10000$ and then converges more slowly to a lower average regret.

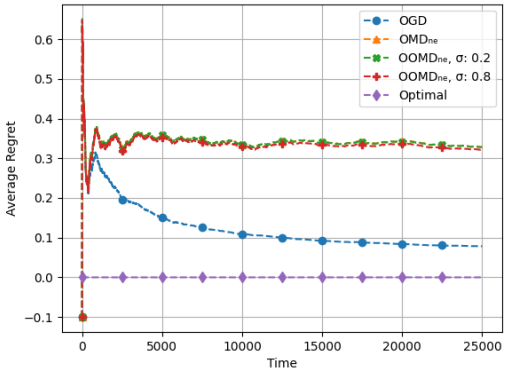


Figure 7: The average regret over the MovieLens request trace.

5 Responsible Research

Since caching is concerned with the moving and storing of data it is wise to handle this data correctly. This data can contain personal or secretive information and should thus be handled accordingly. It is of high importance that one should look at what to and what not to cache as well as how the data is stored. Furthermore, in networks where data is moved around to different servers and caches, it is also crucial that no personal or secretive data is stored at vulnerable locations. One way of making sure no significant accidents occur is to label data based on its risk level. This could then be taken into account in the cost function of OCO policies. What should be done as well is to regularly oversee for example the allocation of data to not overly rely on the learning-based policies.

The reproducibility of this research and the implementation of its algorithm can be low given the complexity of the problem and the solution. All the code with the implementation and experiments can be found on GitHub [2]. This combined with the explanation and notation given in this paper should be sufficient into grasping the problem and solution.

6 Discussion

From each figure can it be seen that an optimistic policy with a high amount of correct recommendations performs only

slightly better than the OMD_{ne} policy but is not as significant as hoped. Significant changes in performance might be discovered when the policies are run in more different cache setups. A lot of parameters such as the catalogue size, cache size, learning rate and percentage of correct recommendations can be tweaked. Moreover, more types of traces can be used to see how that will impact the results. An interesting trace could be a real data set of users interacting with an edge cache.

What is very noticeable over almost all traces is that OGD performs better than OMD_{ne} and $OOMD_{ne}$. This could be due to the significant difference in optimal learning rates. From figure 3 (b) and (c) can the impact of a higher learning rate be seen for these policies. Even though large performance differences can be seen with these higher learning rates it was chosen to go forward with the optimal learning rate because it had been proven to always have a sub-linear regret. More tests for these learning rates could be done under different traces, to see if this significant change in performance would persist.

7 Conclusions and Future Work

An optimistic online caching policy is designed using a negative entropy optimistic online mirror descent algorithm. From the obtained results can it be concluded that the optimistic policy with mostly wrong recommendations has an equal performance in terms of regret as its non-optimistic counterpart. When the policy receives a high percentage of correct predictions then the policy has a lower regret that its non-optimistic counterpart.

Some possible extensions that could be experimented with in future works would be to see how the policy will perform under batch file requests instead of single file requests. The optimality of the proposed optimistic online caching policy could also be tested in a more complex cache network instead of a single cache scenario. Furthermore, in this research were correct and faulty predictions generated according to the quality parameter σ. It can be interesting to see how the integration of the optimistic policy will work with an actual recommender system under real data sets. Lastly, theoretical research could be done to find the optimal learning rates for the two learning rates of the $OOMD_{ne}$ policy since no proofs have been given as of yet for their regrets bounds.

8 Acknowledgements

I would like to sincerely thank: George Iosifidis, Naram Mhaisen and Tareq Salem for helping helped me make this research project come to fruition. I have thoroughly enjoyed every step along the way during this whole process. Thank you for helping me gather the required knowledge to understand this problem and build my interest and desire to expand this research topic. I would also like to thank my research group members: Mikkel Mäkelä and Quentin Oschatz for helping me and doing this research on this topic alongside me.

References

[1] Movielens 20m dataset. https://www.kaggle.com/datasets/grouplens/movielens-20m-dataset/metadata?resource=download, 2018.

[2] G. Admiraal. Optimistic online caching. https://github.com/gadmiraal/optimistic-online-caching, 2022.

[3] A. Beck and M. Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. *Operations Research Letters*, 31(3):167–175, 2003.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] A. Bestavros, R. L. Carter, M. E. Crovella, C. R. Cunha, A. Heddaya, and S. A. Mirdad. Application-level document caching in the internet. pages 166–173, 1995.

[6] S E. Elayoubi and J. Roberts. Performance and Cost Effectiveness of Caching in Mobile Access Networks. In *ICN 2015*, Proceedings of ICN 2015, San Francisco, United States, September 2015.

[7] K. Shanmugam et al. Femtocaching: Wireless content delivery through distributed caching helpers. *IEEE Trans. Inform. Theory*, 59(12), 2013.

[8] R. Fares, B. Romoser, Z. Zong, M. Nijim, and X. Qin. Performance evaluation of traditional caching policies on a large system with petabytes of data. pages 227–234, 06 2012.

[9] S. Geulen, B. Vöcking, and M. Winkler. Regret minimization for online buffering problems using the weighted majority algorithm. volume 17, pages 132–143, 10 2010.

[10] C. A. Gomez-Uribe and N. Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manage. Inf. Syst.*, 6(4), dec 2016.

[11] E. Hazan. Introduction to online convex optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325, 2016.

[12] A. Ioannou and S. Weber. A survey of caching policies and forwarding mechanisms in information-centric networking. *IEEE Communications Surveys & Tutorials*, 18(4):2847–2886, 2016.

[13] P. R. Jelenković and X. Kang. Characterizing the miss sequence of the lru cache. *SIGMETRICS Perform. Eval. Rev.*, 36(2):119–121, aug 2008.

[14] S. Kamali and A. López-Ortiz. *A Survey of Algorithms and Models for List Update*, pages 251–266. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[15] K. C. Kiwiel. Proximal minimization methods with generalized bregman functions. *SIAM J. Control Optim.*, 35(4):1142–1168, jul 1997.

[16] M. Leconte, G. Paschos, L. Gkatzikis, M. Draief, S. Vassilaras, and S. Chouvardas. Placing dynamic content in caches with small population. pages 1–9, 04 2016.

[17] J. S. Liptay. Structural aspects of the system/360 model 85, ii: The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[18] H. B. McMahan. A survey of algorithms and analysis for adaptive online learning. *Journal of Machine Learning Research*, 18(90):1–50, 2017.

[19] N. Mhaisen, G. Iosifidis, and D. Leith. Online caching with optimistic learning, 2022.

[20] G. S. Paschos, A. Destounis, L. Vigneri, and G. Iosifidis. Learning to cache with no regrets. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 235–243, 2019.

[21] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, dec 2003.

[22] A. Rakhlin and K. Sridharan. Online learning with predictable sequences. *Journal of Machine Learning Research*, 30, 08 2012.

[23] T. Salem. Online cache. https://github.com/neu-spiral/OnlineCache/blob/main/tweak_traces.ipynb, 2022.

[24] T. S. Salem, G. Neglia, and S. Ioannidis. No-regret caching via online mirror descent. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[25] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini. Temporal locality in today's content caching: Why it matters and how to model it. *SIGCOMM Comput. Commun. Rev.*, 43(5):5–12, nov 2013.