Optimistic Discrete Caching with Switching Costs

Machine Learning Algorithms for Caching Systems

Lucian Tosa
Supervisors: Georgios Iosifidis, Naram Mhaisen, Fatih Aslan
EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Lucian Tosa
Final project course: CSE3000 Research Project
Thesis committee: Georgios Iosifidis, Naram Mhaisen, Fatih Aslan, Neil Yorke-Smith

Abstract

This paper investigates strategies to limit the cost of switching the cache in the context of an optimistic discrete caching problem. We have chosen as starting point the current state-of-the-art in optimistic discrete caching, the Optimistic Follow-The-Perturbed-Leader (OFTPL) algorithm. We propose two strategies that attempt to limit this cost. The first approach sets a lower bound for the perturbation factor, thus including a mandatory amount of noise to the optimization of the cache state. The second approach investigates dynamically updating the perturbation factor to increases in the switching cost, hence allowing the algorithm to adapt to the scenario at hand. Therefore, to the best of our knowledge, we design the first optimistic discrete caching algorithm that attempts optimizing the switching cost. Finally, we experiment and evaluate our solutions while highlighting their strengths and limitations.

1 Introduction

Caching systems are present in most services and applications we use daily: from watching movies and visiting websites in content delivery networks [1], communication systems [2], wireless content delivery [3] and CPU caching to even the logistics of inventory management for delivery companies. The implementation of caching enables lower latency for customers or users, valuable network efficiency, and enhanced scalability [4]. Due to increasing usage of the aforementioned applications, these properties represent the reason why developing effective caching algorithms is essential to sustainability, cost and time savings.

1.1 Motivation

The goal of this project is to design and evaluate online learning techniques for robust caching systems with real-world value. Common approaches to caching systems are the well-known eviction policies: Least-Recently-Used (LRU), Least-Frequently-Used (LFU), First-In-First-Out (FIFO). These policies employ a strategy to evict an item from the cache when it becomes full in response to a request of an non-cached file. In this case, the policies attempt to add the new item in the cache, but since it is full, a decision needs to be made as to which item to remove. This is where these policies dictate which item will be evicted. Although proven and still frequently used in caching systems today, the existence of time-varying trends in different files, such as a new TV show, can lead to suboptimal performance [5].

Therefore, a common challenge in these systems is to design an online policy that decides dynamically which items to store in cache, without knowing the future requests, so as to maximize the cache hits. Previous work [6] expanded on online learning algorithms using Online Gradient Ascent (OGA) while considering a caching system for a continuous environment.

With the increased development effort towards machine learning algorithms, the usage of predictors for future requests has been a topic of interest. Previous work [7] included implementation and performance guarantees of different online learning algorithms for discrete caching. The algorithms in this paper represent the current state-of-the-art with the help of a prediction oracle of unknown accuracy, thus leading to the term *optimistic*. While having knowledge of future requests makes these algorithms obsolete, having a predictor with high accuracy will inevitably obtain good results. However, the benefit of the Optimistic Follow-The-Perturbed-Leader is that it makes no assumptions on the accuracy of the

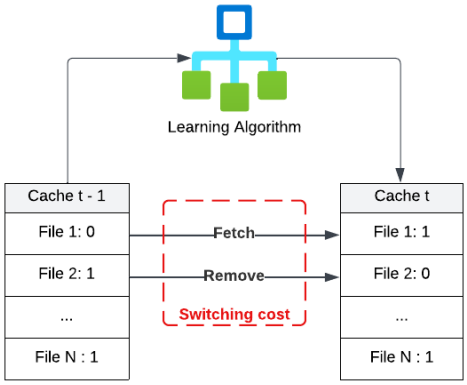


Figure 1: Simplified view of switching cost

predictor, therefore being able to adapt to cases where its accuracy is degrading. As the authors have proved in this work, this algorithm achieves higher performance that many other algorithms, including other optimistic implementations. For this reason, this represents a good candidate to further expand upon.

To be able to evaluate a caching policy, a metric has been defined, namely the *regret*. The works cited above use this metric as a way to evaluate the implemented caching policy against a theoretical "best-in-hindsight" static policy, which assumes complete knowledge of future requests. In essence, this metric evaluates the cache hits that the caching policy is able to obtain.

We can also consider a different metric for evaluating our caching policies, namely the *switching cost*. This cost can be interpreted as penalties incurred by the caching algorithm for swapping the items stored in the cache, and can be visualised in Figure 1. In real-world scenarios this can be seen as replacing items in the warehouse. This example has very high costs, as the old items need to be shipped out and the new ones need to be brought in, therefore it incurs transportation costs, fuel, employees, etc. Another example is changing the movies stored in a caching server. In this case, the switching costs are considerably lower: removing a movie, downloading a new one, and the power used by this process.

While the regret, which considers the number of cache hits of a policy is an informative metric, the costs incurred by switching items in the cache could outweigh the benefits of many cache hits. However, the research cited above does not implement in its optimizations the concept of switching cost. For this reason, the purpose of this research is to investigate the switching cost in an optimistic caching algorithm and evaluate the trade-off between cache hits and switching costs.

1.2 Contribution

We can categorize caching systems as discrete or continuous, where discrete signifies having atomic items that can not be split, and continuous represents items that can be cached partially. In this project we are considering only discrete caching algorithms, but these approaches can be converted to approximate continuous environments as well, by splitting items in chunks and treating it as a discrete problem.

In this paper, we will dive deeper into the Optimistic Follow-The-Perturbed-Leader (OFTPL) algorithm [7] and augment it with strategies for limiting the switching cost. The research questions we are attempting to answer in this work are:

- Under what scenario would OFTPL change the cache state at every step?
- How can the OFTPL algorithm incorporate the switching cost?
- What strategy can be implemented to limit this switching cost?

To this end, we started by recreating the algorithm in [7] and evaluating it on specific request patterns and on different predictor accuracies. As expected, the algorithm performs well in terms of regret regardless of the accuracy of the predictor. However, we have also evaluated the algorithm in terms of the switching cost. Under specific conditions, we have observed that a perfect predictor will lead to a linearly growing switching cost. A perfect predictor will influence the perturbation factor of the algorithm, which has the purpose of introducing noise inversely proportional to the accuracy of the predictions. This paper will investigate heuristic approaches to limiting the switching cost by adapting the perturbation factor in 2 different ways: by limiting the perturbation factor above a threshold, and by dynamically adapting it with the switching cost, and not only with the predictor's accuracy.

2 Methodology and Background

2.1 Related work

The caching problem has always been prevalent and research in this field has been continuously growing, as the survey [8] presents a number of recent developments. Therefore, the current landscape contains a variety of caching algorithms. There exists the classic caching strategies based on eviction policies, such as LFU and LRU, and modern variations of these [9, 10].

A significant body of research has focused on the development of online learning algorithms, such as Online Gradient Descent [6] for continous caching or Follow-The-Perturbed-Leader [11] for discrete caching. However, increasing work in the field of machine learning has also enabled the development of optimistic learning, by using predictions for future requests as a tool to compute the best caching policy [12]. While this work has taken into account the predictions' accuracy, further work [13] has developed a method of using untrusted predictions, therefore being agnostic to the predictor's accuracy, achieving *competitive-ratio* guarantees. Furthermore, [7] has extended this approach to discrete caching with untrusted predictions, by developing the Optimistic Follow-The-Perturbed-Leader algorithm, which is the focus of this paper. This algorithm has been shown to achieve sub-linear regret regardless of the predictor's performance.

The purpose of this paper is to extend the work done in [7] by augmenting the OFTPL algorithm while considering the switching cost and striking a balance between this and the regret metric. Finally, it is important mentioning that including the switching cost in the optimization function is not a new concept. There have been online caching algorithms which consider this cost [14]. However, while this work is studying the similar Follow-The-Perturbed-Leader (FTPL) algorithm, it is not the optimistic variant that we are investigating, therefore not using a predictor for future request traces. The use of a predictor is an important development that leads to performance gains over the plain FTPL algorithm. However, as we will discover in the later sections, it can also represent a detriment for the

other metric we are evaluating, the switching cost. Therefore, the solutions discussed in this paper can't be applied to the non-optimistic variant directly.

2.2 Methodology

For the purpose of this paper, we have employed an experimental research design to evaluate and augment the OFTPL caching algorithm against various request patterns and prediction accuracies. This choice has been done to provide empirical evidence of current performance considering the switching cost metric and the performance of our improvements.

The first step in investigating strategies of limiting the switching cost was to evaluate the current state-of-the-art in terms of this metric. By recreating the Optimistic Follow-The-Perturbed-Leader algorithm in [7] we could observe its performance on custom-defined request patterns that may not necessarily appear in real-world applications. One such request pattern that we observed to produce surprising results is the round-robin trace, which requests files consecutively at every time step. To evaluate the OFTPL algorithm, a predictor must also be used. This has been achieved by using the known request in advance and predicting with a specified accuracy. The request patterns and predictors used will be described in more detail in subsection 3.2.

Using the round-robin request pattern, and a predictor with 100% accuracy [1], and for certain sizes libraries and caches, the caching algorithm will attempt to switch the item in the cache at every time step to accommodate for the perfect predictions. Having a perfect predictor will cause the OFTPL algorithm to completely trust the predictor and not introduce any noise for the purpose of exploration. The noise is caused by the perturbation factor, which is inversely proportional to the predictor's accuracy. Therefore, we see that in this scenario, the perturbation factor will go to 0. In this case, the caching policy does indeed achieve great performance in terms of cache hits by having negative regret, but we also identify that the switching cost is linearly growing with time, since it modifies the cache state by 2 items at every time step.

Thus we can observe that there is a trade-off to be made between achieving the largest number of cache hits, and having a low switching cost. And we can also notice that the balance of the two stands in the perturbation factor.

To obtain our results, we have pursued 2 different heuristic-based approaches of modifying the perturbation factor, in the hopes of lowering the switching cost. Since having a perturbation factor equal to 0 causes high switching costs, the first approach was to limit this factor above a threshold, thus, never allowing the perturbation to hit 0. The second approach was to dynamically adapt this variable with the switching cost, by integrating this cost in the calculation of the perturbation factor. This is a more advanced approach than a threshold as we have observed that different scenarios require different thresholds.

Using these heuristics we have run new experiments to evaluate the switching cost, and we have observed that for some scenarios, the switching cost can be decreased significantly, but at the cost on an increased regret.

[1] which would be very desirable in real-world scenarios

3 System Model and Problem Statement

3.1 Model Preliminaries

Cache. The cache represents a subset of the entire library of N items. The cache has a maximum size of C items, where usually, $C < 10\% \cdot N$. The cache is considered a high-speed storage layer that holds only C items which are a copy of the total library of items, that is stored in a slower storage. Caches are used to reduce the time required to access the items and to reduce the load on the main storage space. A cache can be encoded as a 1-hot vector $y_t = \{0,1\}^N, \sum_{i=1}^{N} y_i \leq C$ with value 1 for the item that is cached. We define the set of valid caching states

$$\mathcal{X} = \left\{ y_t \in \{0,1\}^N \middle| \sum_{i=1}^{N} y_i \leq C \right\}$$

Requests. Our caching system is slotted in discrete time steps, $t = 1, 2, ..., T$. The system receives a request at every time step for a specific item from a library of total N items. The request is encoded as a 1-hot vector, where only 1 file is requested per time slot

$$\theta_t = \{0,1\}^N, \sum_{i=1}^{N} \theta_i = 1$$

Predictions. Since we are developing an optimistic caching algorithm, be also receive predictions for the next request, which are encoded similarly to the requests,

$$\tilde{\theta}_t = \{0,1\}^N, \sum_{i=1}^{N} \theta_i = 1$$

Gradient. We denote the accumulated sum of all the requests until time t as

$$\Theta_t = \sum_{i=1}^{t} \theta_i$$

Cache hit. A cache hit occurs when the item requested at at a time step is present in the caching state at that time. This results in a faster response.

Cache miss. A cache miss occurs when the item requested at a time step is not present in the cache state. In this case, the item will be retrieved from the main library incurring a latency cost.

Perturbation factor. The perturbation factor is a term used in the OFTPL algorithm in [7] that increases with the error of the predictions. It is used in the OFTPL algorithm as a way to add noise in case the predictions are not accurate.

$$\eta_t = \frac{1.3}{\sqrt{C}} \left(\frac{1}{\ln(Ne/C)} \right)^{\frac{1}{4}} \sqrt{\sum_{\tau=1}^{t-1} \left\| \theta_\tau - \tilde{\theta}_{\tau-1} \right\|_1^2}$$

3.2 Problem Statement

Predictions. There is a prediction oracle that, at each time step, is able to provide a prediction of unknown accuracy of the next request. Using the predictions provided by the

oracle we compute our utility function to determine the next caching state, before we receive the request. Predictions are also encoded as 1-hot vectors $\tilde{\theta}_t = \{0,1\}^N$.

Benchmark. To be able to evaluate the performance of our caching policy, we need to define what is the optimal caching policy. The optimal policy we choose is the "best-in-hindsight" $x^\star$, which is chosen assuming access to all future requests. Thus, the "best-in-hindsight" policy becomes the static policy which obtains the maximum number of cache hits. A static policy is one such that it keeps the same cache state for every time step.

Regret. To evaluate our online learning policy, we use the static regret metric which compares our policy to the hypothetical solution $x^\star$.

$$R_T(\{x\}_T) = \sup_{\{f_t\}_{t=1}^T} \left\{ \sum_{t=1}^{T} f_t(x^\star) - \sum_{t=1}^{T} f_t(x_t) \right\}.$$

Switching cost. To evaluate the incurred penalty of switching the cache, we use the switching cost [14] as the accumulated sum of the cache state changes at every time step. The factor D represents the weight of the switching cost and it can be adapted to the scenario in which it is used.

$$S_T(\{x\}_T) = \frac{D}{2} \sum_{t=2}^{T} \|y_t - y_{t-1}\|_1$$

Switching regret. To evaluate both the regret metric and the switching cost in the same formula, we use the so-called switching regret [14] defined as the sum of the regret and the switching cost.

$$SR_T(\{x\}_T) = R_T + S_T$$

4 Limiting the switching cost in OFTPL

To fully understand how the switching cost grows in the OFTPL algorithm, we have evaluated the current state-of-the-art in [7] with different request patterns, and different predictors. These experiments will be shown and described in more detail in the next section. However, the scenario which caused surprising results is the round-robin request pattern coupled with a perfect predictor. While this scenario is very rare in real-world application, even arguably impossible to achieve, it is a good example of the worst existing scenario that can cause an extreme increase of the switching cost.

In the aforementioned scenario, due to the perfect predictions, the algorithm will set the perturbation factor to 0, thus focusing solely on exploitation rather than exploration. In this case, the caching state will be decided entirely by the past requests, and the prediction. While this is a perfect scenario that achieves 100% cache hit rate, it can be observed in Figure 2a, that the switching cost increases linearly with time. This is due to the fact that the caching policy will always cache the next file, and remove a file from its cache.

With this observation we can deduce that there is a balance between the cache hits, measured by the regret, and the switching cost. It also seems that we can control this balance by adapting the perturbation factor. To be able to evaluate the balance between the regret metric and the switching cost, we will also evaluate our algorithms against the switching regret, defined previously. In the following subsections we will discuss 2 approaches that should help control the balance between the regret and the switching cost.

4.1 Bounding the perturbation factor

Algorithm 1 OFTPL with bounded perturbation factor

Input: $\Theta_0 \leftarrow \{0\}^N, L$
Output: $\{y_t \in \mathcal{X}\}_T$ ▷ Discrete caching vector at each time step
$\gamma \sim \mathcal{N}(0, 1_{N\times 1})$ ▷ Sample perturbation vector
for t = 1, 2, ... **do**
  $\tilde{\theta}_t \leftarrow$ `prediction` ▷ Receive prediction for next request
  $\eta_t \leftarrow \frac{1.3}{\sqrt{C}} \left(\frac{1}{\ln(Ne/C)}\right)^{\frac{1}{4}} \sqrt{\sum_{\tau=1}^{t-1} \left\|\theta_\tau - \tilde{\theta}_{\tau-1}\right\|_1^2}$ ▷ Update perturbation
  $\eta_t \leftarrow \max\{L, \eta_t\}$ ▷ Apply threshold
  $y_t \leftarrow argmax_{y\in\mathcal{X}} \left\langle y, \Theta_{t-1} + \tilde{\theta}_t + \eta_t\gamma \right\rangle$ ▷ Calculate caching vector
  $\Theta_t \leftarrow \Theta_{t-1} + \theta_t$ ▷ Receive the request and update gradient
end for

Given the observation that a perturbation factor equal to 0 will cause high switching costs, the first approach tested was to limit the perturbation factor above a preset threshold, noted by L. Thus, by imposing this threshold, the algorithm will never completely eliminate the exploration factor. In this case, at every time step, the algorithm calculates the perturbation factor in the same way as in [7], but after calculation, a minimum threshold defined by the user is imposed. In this case, the choice for the caching state will introduce noise, or perturbation, regardless of the accuracy of the predictor.

With the introduction of the threshold, a new challenge appears. What is the optimal value for L that achieves the best switching regret? Through experiments that will be further described in the next section, we observed that this variable is not a one-size-fits-all. Different request patterns and predictor accuracies will require a different threshold to achieve better results compared to having no threshold. Furthermore, we have observed that in more common cases, when the predictor does not have 100% accuracy, the perturbation factor does not go to 0 and increases with time, therefore the threshold does not affect the outcome after some time horizon in these cases.

However, for the scenarios under test, we have identified that setting very high thresholds will obtain very low switching regret. While this seems like a good result, the algorithm is essentially not learning. Since the perturbation factor is so high, the cache state will always be skewed to the random vector chosen at the beginning, therefore incurring very low switching cost and performing badly in terms of cache hits. Setting the threshold to lower values can achieve $\approx$ 55% decrease in switching regret, with a $\approx$ 66% decrease in switching cost, while maintaining negative regret, as seen in Figure 2b.

With these results considered, we identified that the main issue with using a threshold for the perturbation factor is the fact that it requires manual tuning to the request pattern, the predictor accuracy, information that is not known in real-world applications, and of course that it does not adapt over time. Considering these observations, we move on to the second attempted approach.

4.2 Switching cost informed perturbation factor

From the previous discussion, it has become apparent that the perturbation factor should dynamically adapt to the scenario at hand. While it is already dynamically updated with regards to the predictor's accuracy, it does not optimize for the switching cost as well. Thus to optimize the algorithm for the switching costs, we need to modify the update step of the perturbation factor.

We observed that a low perturbation factor increases the switching cost, and a high perturbation factor decreases it. Therefore, one option is to change this term so that it increases proportionally to the switching cost. This choice is reasonable, since high switching costs will lead to a higher perturbation factor, thus indirectly decreasing the weight of the prediction in line 7 of Algorithm 2. To obtain this behavior for the perturbation factor, we have included the accumulated switching cost under the root next to the accumulated prediction errors, as in line 6 of the following algorithm.

Algorithm 2 OFTPL with switching cost informed perturbations

1: **Input:** $S_0 \leftarrow 0, \Theta_0 \leftarrow \{0\}^N, D$
2: **Output:** $\{y_t \in \mathcal{X}\}_T$ ▷ Discrete caching vector at each time step
3: $\gamma \sim \mathcal{N}(0, 1_{N\times 1})$ ▷ Sample perturbation vector
4: **for** t = 1, 2, ... **do**
5: $\tilde{\theta}_t \leftarrow$ `prediction` ▷ Receive prediction for next request
6: $\eta_t = \frac{1.3}{\sqrt{C}} \left(\frac{1}{\ln(Ne/C)}\right)^{\frac{1}{4}} \sqrt{\sum_{\tau=1}^{t-1} \left\|\theta_\tau - \tilde{\theta}_{\tau-1}\right\|_1^2 + S_{t-1}}$ ▷ Update perturbation
7: $y_t \leftarrow argmax_{y\in\mathcal{X}} \left\langle y, \Theta_{t-1} + \tilde{\theta}_t + \eta_t\gamma \right\rangle$ ▷ Calculate caching vector
8: $\Theta_t \leftarrow \Theta_{t-1} + \theta_t$ ▷ Receive the request and update gradient
9: $S_t \leftarrow S_{t-1} + \frac{D}{2} \|y_t - y_{t-1}\|_1$ ▷ Update the accumulated switching cost
10: **end for**

The introduction of the switching cost under the root has been made heuristically. This choice is reasonable considering that the perturbation factor contains the accumulated prediction errors under the root as well. Therefore, this allows us to scale the perturbation factor with both the predictor's accuracy and the switching cost. We note that a theoretical analysis is not in the scope of this paper, and is left as a suggestion for future work.

Besides the addition to the update step of the perturbation factor, the algorithm also needs to keep track of the accumulated switching cost, its formula being described in the previous section. This step requires an additional variable D, but in this case, this variable does not need to be adapted. The parameter D dictates how *expensive* is the cost of switching the cache. As exemplified in the introduction, a CDN server that keeps movies and websited would have low cost for deleting and adding new items in the cache. Whereas, an online store's warehouse that acts as a cache for a neighbouring city has a much higher cost for removing and adding new items, due to the costs of logistics. Therefore, this parameter should be set according to the problem that the caching system aims to solve.

With the new perturbation factor, the algorithm is able to adapt to different request patterns and predictors that cause varying switching cost, as opposed to the first approach which required unavailable knowledge to choose the right value for the threshold.

5 Experimental Setup and Results

In this section we will present the experiments ran on the approaches described previously. We will compare to the OFTPL algorithm described in [7] with no modifications.

The algorithm and experiments have been implemented in Python 3.12 using a Jupyter Notebook. The program was run in Visual Studio Code on a system running macOS 14.5 on an Apple M3 Pro chip.

For part of our data used for evaluation, we have synthetically generated requests according to different patterns. For these request patterns, we have used a time horizon $T = 10000$, a library size $N = 1000$ and a cache size $C = 50$.

The following request patterns have been generated:

- round-robin requests - At every time step the request contains a different file, sequentially. More specifically, at time t, file tN will be requested.
- Zipf requests - Files are requested according to a Zipf [2] distribution, with parameter $a = 2$.
- random requests - At every time step, the request contains a file drawn from a uniform distribution. The experiments for this request pattern have been included in the appendix and can be seen in Figure 5.

Besides synthetic data, we have also generated request traces from the MovieLens dataset [15]. More specifically, the dataset "ml-latest-small" [3] has been chosen, having a total of 100.000 ratings. A request is generated from a timestamped movie rating and are sent to the caching algorithm in order of their timestamp. For computational reasons, we have used only the first 10.000 requests, therefore leading to a time horizon $T = 10000$. For this dataset we identify 3218 unique movies that have been rated at least once, leading to setting the library size $N = 3218$, and we set the cache size $C = 50$, around 1.5% of N.

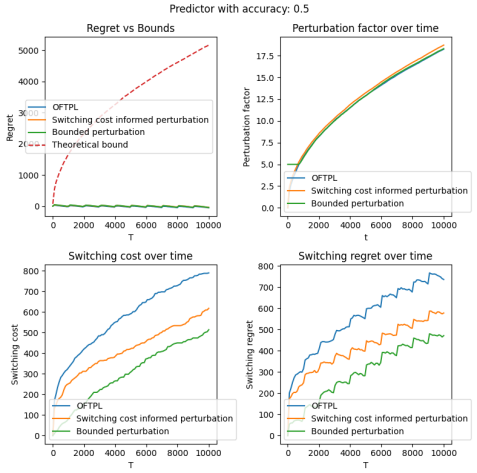
Since we are constructing an optimistic caching algorithm, we also designed a predictor that has different accuracies. For every request trace, we have used a synthetic predictor as also used for the experiments in [7]. The predictions have a α probability of being correct, namely, at each time step t, we generate a one-hot encoded vector $\tilde{\theta}_t$, which is identical to the request θ_t with probability α, or with probability $1 - \alpha$ we choose any file except the requested one, and set it to 1.

5.1 Round-robin requests

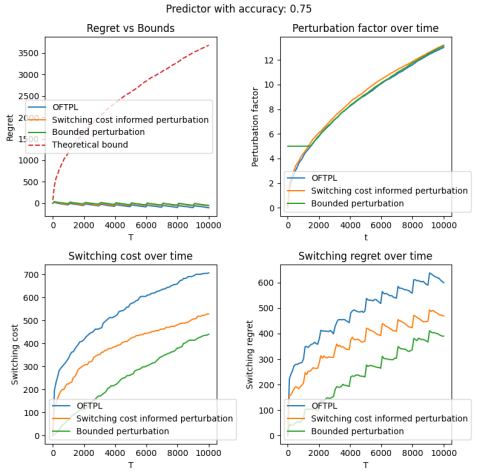
The first experiment we will discuss involves evaluating the algorithms under a round-robin request pattern and with a predictor of various accuracies. In Figure 2, we see the plots of the key metrics we used to evaluate our algorithms. Here we can see how the 3 implementations: the unmodified OFTPL from [7], the OFTPL with a limited threshold from subsection 4.1, and the OFTPL with a switching cost informed perturbation factor from subsection 4.2. In this experiment we observed the linearly growing switching cost in sub-figure 2d, which was caused by a predictor with 100% accuracy, therefore leading to a perturbation factor equal to 0. This experiment is what led us to the 2 approaches previously described. To evaluate the proposed solutions, we set the threshold in the first approach to $L = 5$ for all experiments, which was chosen by trial and error. For the second proposed solution we

[2] https://en.wikipedia.org/wiki/Zipf%27s_law

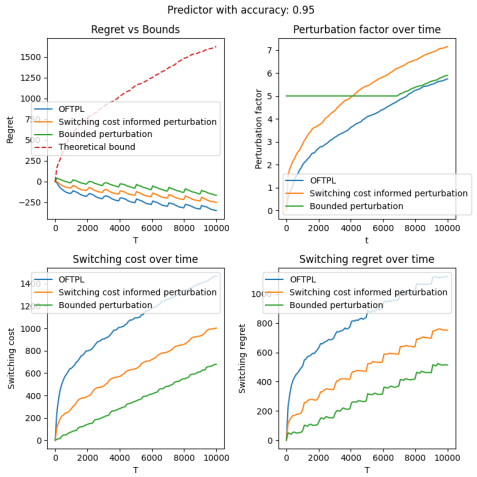
[3] https://grouplens.org/datasets/movielens/latest/



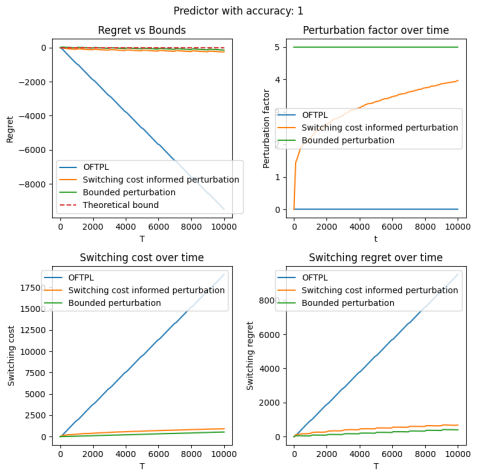
(a) Predictor with accuracy 0.5



(b) Predictor with accuracy 0.75



(c) Predictor with accuracy 0.95



(d) Predictor with accuracy 1

Figure 2: Plots showing metrics of the OFTPL algorithm under round-robin requests and using a predictor with various accuracies. In each sub-figure, 4 plots are shown: top-left represents the regret metric, top-right represents the perturbation factor, bottom-left represents the switching cost, and bottom-right, the switching regret.

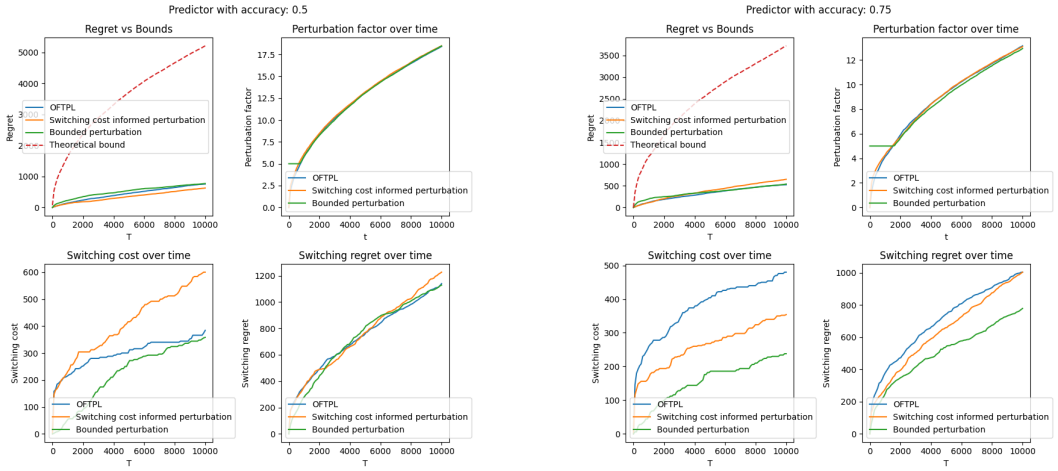
chose $D = 2$, therefore giving a unit penalty, to ensure fairness between experiments. In this experiment we saw the biggest improvements to the switching regret when the predictor's accuracy was set to 1, therefore achieving a 96% decrease with the first approach and a 91% decrease with the second approach, and while the regret is closer to 0, it is still negative while achieving very low switching costs. For the other preset accuracies of the predictor, we see a similar trend with improvements ranging from 30% to 60% in switching regret.

Observation 1. An increase in the predictor's accuracy will always lead to a lower perturbation factor, which, depending on the request pattern, may lead to higher switching

costs. At the same time, higher accuracies will lead to lower regret values.

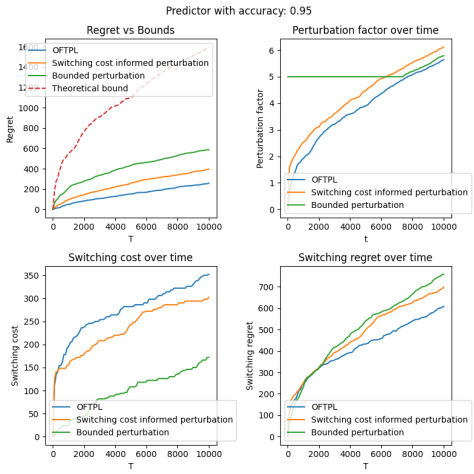
Observation 2. Higher values for the perturbation factor will lead to lower switching cost, but a higher regret metric. For the chosen threshold $L = 5$, which is a high value for this specific scenario, Algorithm 1 achieves lower switching regret and switching cost then Algorithm 2, but at the expense of regret.

5.2 Zipf requests

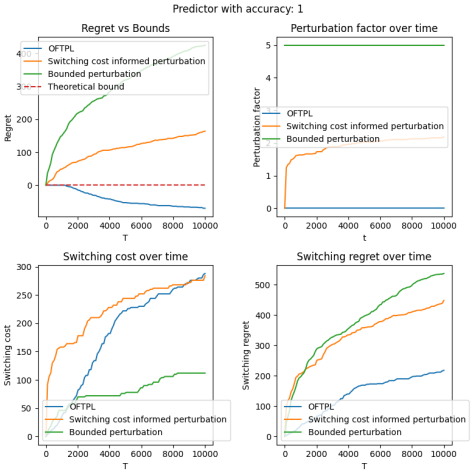


(a) Predictor with accuracy 0.5

(b) Predictor with accuracy 0.75



(c) Predictor with accuracy 0.95



(d) Predictor with accuracy 1

Figure 3: Plots showing metrics of the OFTPL algorithm under Zipf requests and using a predictor with various accuracies.

The second experiment follows the same template as the first. Here we have evaluated the three algorithms against the metrics already discussed while receiving requests that

follow the Zipf distribution. Results can be found in Figure 3.

Observation 1. This distribution behaves completely different as the one from experiment 1. Here we observe that the switching cost of the OFTPL algorithm converges to the same value regardless of the predictor's accuracy, whereas in experiment 1, the switching cost increased with the accuracy of the predictor.

Observation 2. Due to the nature of a Zipf distribution, the algorithm does not incur high switching costs, which causes both solutions to perform similar or worse than the normal OFTPL algorithm in terms of switching regret. Because of the highly localized requests, the perturbation factor introduces unwanted noise, leading to introducing higher switching costs or cache misses. Therefore, we can observe that highly localized request patterns would not benefit from the proposed solutions.

5.3 MovieLens dataset

For the third experiment we have used request traces drawn from a MovieLens dataset. This process has been described in more detail in the introduction of this section. Figure 4 present the plots in the same manner as the other two experiments, by comparing the three algorithms in the key metrics we have defined.

Observation 1. Real-world request traces lead to more evident performance benefits. We observe that the two proposed solution achieve better switching regret, switching costs, and a regret that is still withing bounds, for accuracies 0.5, 0.75, and 0.95. Here we observe a decrease in switching regret of up to 50% for Algorithm 1, and up to 31% for Algorithm 2.

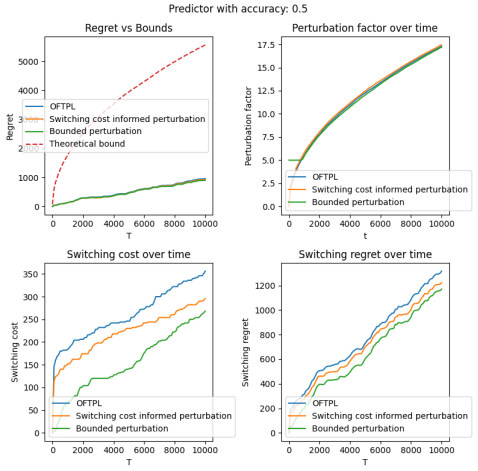
Observation 2. In this case we see again that Algorithm 1 performs better than Algorithm 2, with the same well-chosen value for $L = 5$. However, while it performs better in switching regret and switching costs, it incurs a higher regret, especially for a prediction accuracy of 1, therefore pointing to a clear trade-off between the 2 methods.

6 Responsible Research

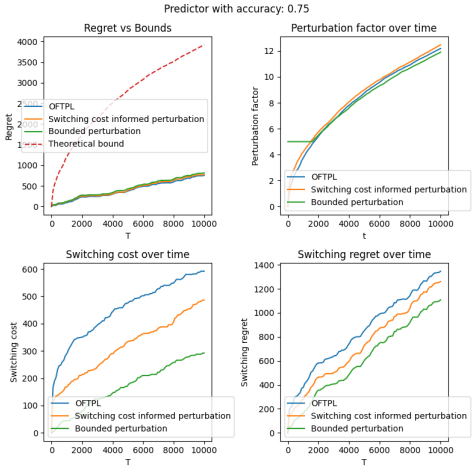
The ethical considerations of our research, as well as the reproducibility and transparency of our methods and experiments should be discussed to ensure a responsible research process. This requires a discussion on the data we have chosen and the experiments we have shown.

Throughout our research we have used either synthetically generated data, or publicly available datasets which do not contain personally identifiable information. For the synthetic data we have used, we have described the methods of generating it, ensuring reproducibility. For the MovieLens dataset, we have mentioned the exact version of the dataset we have chosen, how we generated the requests from the ratings data and how we selected the requests to use.

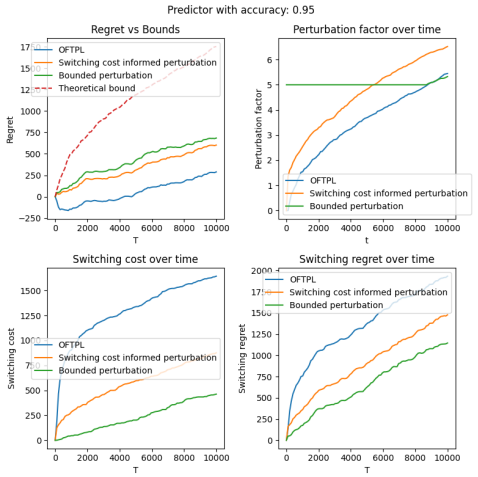
To ensure transparency and reproducibility of our work, we have given pseudocode for both of our proposed solutions and have described their inner workings in detail. In terms of experiments, we have clearly and transparently shown all of our results in the form of plots and formed an objective interpretation of the figures. The metrics we have chosen to plot have been described mathematically in Section 3.



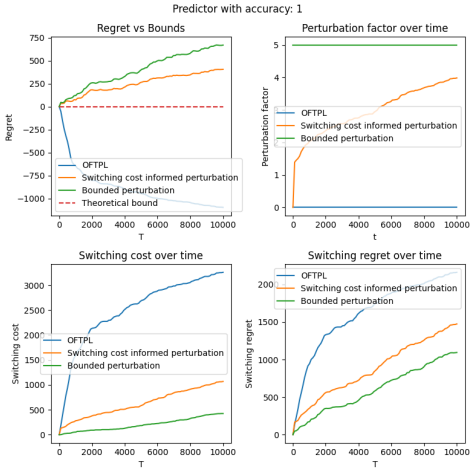
(a) Predictor with accuracy 0.5



(b) Predictor with accuracy 0.75



(c) Predictor with accuracy 0.95



(d) Predictor with accuracy 1

Figure 4: Plots showing metrics of the OFTPL algorithm under requests generated from the MovieLens dataset and using a predictor with various accuracies.

7 Conclusions and Future Work

In summary, the main research question we aimed to answer through this research was "What strategy can be implemented to limit the switching cost?". In this project we focused on developing an optimistic online learning algorithm for discrete caching that also considers the switching cost as an optimization target. We have developed on the OFTPL algorithm presented in [7] and proposed two solutions for limiting the switching cost incurred by this caching policy. The first proposed solution involves bounding the perturbation factor by a preset value, and the second solution updates the calculation of the perturbation factor to take into account the accumulated switching cost alongside prediction errors. We have

presented experiments for different request patterns and prediction accuracies in which we compared the proposed solutions to the original algorithm. As a result of our experiment we have identified that in the MovieLens dataset, the proposed solutions perform better than the initial OFTPL algorithm in terms of switching cost and switching regret while still maintaining the regret within the theoretical bounds. We have also identified a scenario in which these two approaches, namely requests that follow a Zipf distribution. While comparing the two solutions with each other, we have noticed that the more simple bounded perturbation performed better in some cases than the switching cost informed perturbation. This does come at the cost of an increased regret, therefore there is a clear trade-off to be made between the 2 approaches. The significant disadvantage to the first approach is that it requires setting an additional variable L. To choose the optimal value for this variable requires knowledge of the scenario in which the algorithm is applied, information that is possibly not known. However, it can be heuristically chosen, as we have done in our experiments. In contrast, the second approach represents an online learning method as it is able to adapt to different request patterns and prediction accuracies on the fly.

There are a number of possible improvements and suggestions for future research in this topic. For the moment the variable D is fixed in time and represents the weight of the switching cost. However, in real-world applications, the cost of switching the cache can vary with time, for example, internet speeds can vary in a caching server for movies. On the same topic, we have considered that every file/item has the same weight, or the same cost of being removed or added to the cache. But, by using the same example, different movies have different file sizes, therefore incurring a different cost to bring into the cache. Furthermore, we have made the assumption that the cost of adding an item into the cache is the same as removing one. This is not always true, as the time it takes to download a file is often higher than the time it takes to remove one from the cache. Additionally, the second approach we have proposed only uses the past caching states for calculating the switching cost. However, as we use predictions for the next request, implementing optimism in the calculation of this sum can also be further investigated. Finally, the solutions we have proposed have been chosen heuristically and we have only verified their performance experimentally. Thus, a theoretical analysis of these chosen solutions, and their performance guarantees would represent a valuable addition to this research field.

References

[1] T. Bektas, O. Oguz, and I. Ouveysi, "Designing cost-effective content distribution networks," *Computers Operations Research*, vol. 34, no. 8, pp. 2436–2449, 2007.

[2] G. S. Paschos, G. Iosifidis, M. Tao, D. Towsley, and G. Caire, "The role of caching in future communication systems and networks," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 6, pp. 1111–1125, 2018.

[3] K. Shanmugam, N. Golrezaei, A. G. Dimakis, A. F. Molisch, and G. Caire, "Femtocaching: Wireless content delivery through distributed caching helpers," *IEEE Transactions on Information Theory*, vol. 59, no. 12, pp. 8402–8413, 2013.

[4] A. Datta, K. Dutta, H. Thomas, and D. VanderMeer, "World wide wait: A study of internet scalability and cache-based approaches to alleviate it," *Management Science*, vol. 49, no. 10, pp. 1425–1444, 2003.

[5] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," 2016.

[6] G. S. Paschos, A. Destounis, L. Vigneri, and G. Iosifidis, "Learning to cache with no regrets," 2019.

[7] N. Mhaisen, A. Sinha, G. Paschos, and G. Iosifidis, "Optimistic no-regret algorithms for discrete caching," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 6, pp. 1–28, Dec. 2022.

[8] G. Paschos, G. Iosifidis, and G. Caire, "Cache optimization models and algorithms," 2019.

[9] A. Giovanidis and A. Avranas, "Spatial multi-lru: Distributed caching for wireless networks with coverage overlaps," 2016.

[10] E. Leonardi and G. Neglia, "Implicit coordination of caches in small cell networks under unknown popularity profiles," 2018.

[11] R. Bhattacharjee, S. Banerjee, and A. Sinha, "Fundamental limits of online network-caching," 2020.

[12] L. E. Chatzieleftheriou, M. Karaliopoulos, and I. Koutsopoulos, "Jointly optimizing content caching and recommendations in small cell networks," *IEEE Transactions on Mobile Computing*, vol. 18, no. 1, pp. 125–138, 2019.

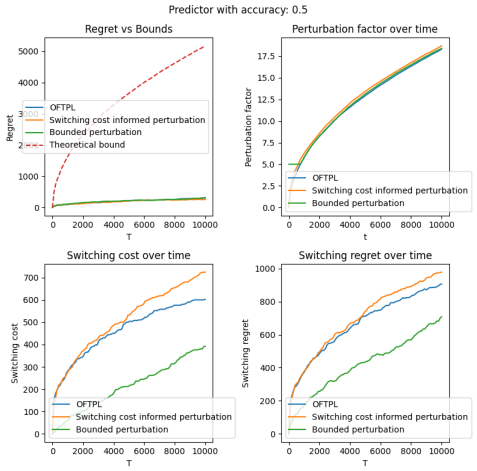
[13] T. Lykouris and S. Vassilvitskii, "Competitive caching with machine learned advice," 2020.

[14] S. Mukhopadhyay and A. Sinha, "Online caching with optimal switching regret," 2021.

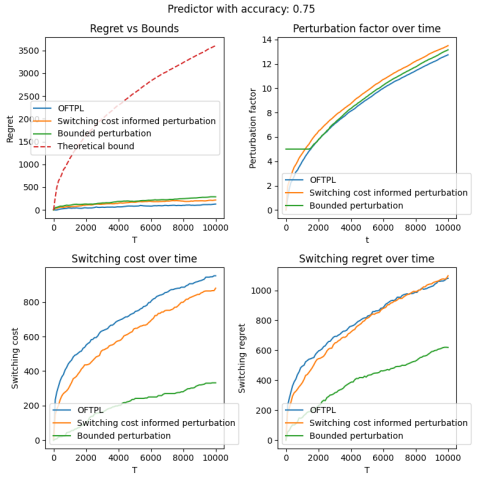
[15] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, vol. 5, dec 2015.

A Additional experiments

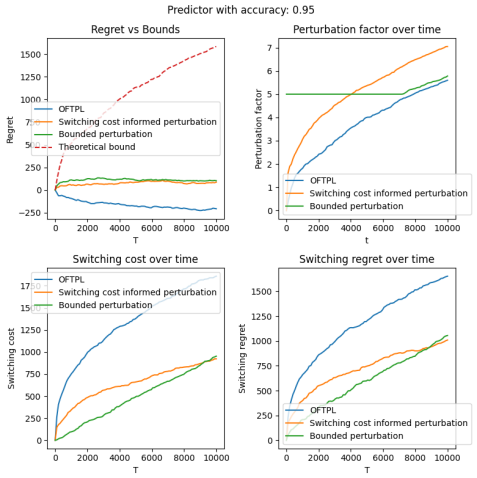
A.1 Uniformly random requests



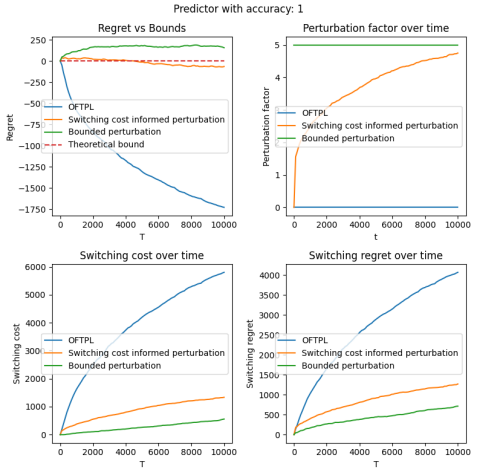
(a) Predictor with accuracy 0.5



(b) Predictor with accuracy 0.75



(c) Predictor with accuracy 0.95



(d) Predictor with accuracy 1

Figure 5: Plots showing metrics of the OFTPL algorithm under uniformly random requests and using a predictor with various accuracies. In each sub-figure, 4 plots are shown: top-left represents the regret metric, top-right represents the perturbation factor, bottom-left represents the switching cost, and bottom-right, the switching regret.