DATA INTELLIGENCE APPLICATIONS
PRICING & ADVERTISING

DATA INTELLIGENCE APPLICATIONS

Andrea Bionda
Department of Computer Science and Engineering
Politecnico di Milano
andrea.bionda@mail.polimi.it

Damiano Derin
Department of Computer Science and Engineering
Politecnico di Milano
damiano.derin@mail.polimi.it

Andrea Diecidue
Department of Computer Science and Engineering
Politecnico di Milano
andrea.diecidue@mail.polimi.it

Antonio Urbano
Department of Computer Science and Engineering
Politecnico di Milano
antonio.urbano@mail.polimi.it

Enrico Voltan
Department of Computer Science and Engineering
Politecnico di Milano
enrico.voltan@mail.polimi.it

November 12, 2020

ABSTRACT

The goal of the project is to model a scenario in which a seller exploits advertising tools in order to attract more and more users to its website, thus increasing the number of possible buyers.

The seller has to learn simultaneously the conversion rate and the number of users the advertising tools can attract.

In this report, we walk through the description of the specific scenario we have studied and the definition of the algorithm design choices we adopted in order to reach our goal. Then the achieved experimental results will be presented via some useful plots and a final conclusion summarizing the whole work.

Contents

Chapter 1

Introduction: Scenario Setup

We have studied a possible real world scenario in which a seller wants to increase the number of possible buyers by exploiting advertising tools.

The product we have considered is a particular pair of shoes which has a production cost (without loss of generality we can assume that the production cost is null) and a sell price.

The analysed campaign consists of three sub-campaigns, each with a different ad to advertise the product and each targeting a different class of users. Each class is defined by the values of pre-defined features.

The feature space we have considered is characterized by two binary features:

- *Age<30* or *Age>30*
- *Profession* that can be either *student* or *worker*

So according to the values of the above described features, we can distinguish among the following classes of users:

- *Elagant*: a worker with age>30
- *Casual*: a student with age<30
- *Sport*: a worker with age<30

Chapter 2

Advertising: Clicks Maximization

In this section we focus on the advertising and more precisely on the clicks maximization problem. The goal of this part is to optimize the budget allocation over the three sub-campaigns in order to maximize the total number of clicks we can get. In particular given the assignment 8 we have designed a combinatorial bandit algorithm for addreassing this task. In the following chapter we are going to:

- describe the environment setup;
- explain the algorithm design choice;
- comment the obtained results.

2.1 Environment

The **Environment** returns the reward associated to each sub-campaign, by iterating over the days for the entire time horizon. In our setting, it computes the optimal number of clicks knowing the real functions, one for each sub-campaign, generating the number of clicks given a bid value ***n(x)***:

$$n(x) = c_M \cdot (v_M - e^{-\alpha x}) \tag{2.1}$$

where:

- c_M: is the maximum number of clicks the considered sub-campaign can reach in a day;
- v_M and α: are values associated to each sub-campaign defining the actual bidding curve.

The Environment, given the index of the bid chosen by the Learner, returns the collection of rewards associated to each sub-campaign in a completely transparant way to the learning algorithm. How the learner works and its implementation will be described in the following section.

2.2 Algorithm Design Choices

This section deals with the description of the algorithm design choices we adopted and it also contains the most relevant reference to the code. In the class *BiddingEnvironment.py*, which extends the class *Environment.py* we have defined the actual environment we are working in, described in section before, with the definition of the bids space and the three curves *n(x)*. In order to find the best budget allocation over the three sub-campaigns able to maximize the total number of clicks, we have designed a combinatorial bandit algorithm.

In the first phase of the algorithm, we learn the model of each sub-campaign from the observation we get. To do that in *GP_Learner.py* we have used a *Gaussian process regressor*. Afterward, the model of each sub-campaign is updated using those observations and the GP will reduce the uncertainty of its estimation. In this way, for each new collected sample, the function estimated by the GP approaches to the real function.

In order to properly used a GP regressor, we had to normalize the data. The bid space was already defined in range [0,1] and so the input variable has not to be normalized. So, we only needed to normalize the target and we have done it in the construction of the gaussian process.
A Gaussian proccess is completely defined by its mean and its covariance.Since we don't have any prior information we assumed to have zero mean and the covariance given by the squared exponential kernel function **k(x,x')**:

$$k(x, x') = \theta^2 e^{-\frac{(x-x')^2}{2l^2}} \tag{2.2}$$

where:

- l: is the *lengthscale*;
- θ: is the *scale factor*

The optimal value of the two hyperparameters have been found by maximiziation of the marginal likelihood during the fit proccess.

In the second phase of the algorithm we have used the values from the learned model to solve the problem of finding the best budget allocation to be set for the current day. In the *Optimizer.py* class we have implemented a modified version of the dynamic programming algorithm used for solving the knapsack problem. More precisely,we have used a matrix in which each row represents the fact that at each step a new sub-campaign enters the problem, while for the columns we have discretized the whole budget in 20 possible uniformly distributed combinations of the budget allocation.

Each cell of the matrix contains the value of the best allocation for the considered row and column. The result is given by the maximization of the sum of the values provided by the best solution of the problem solved in the previous row (i.e. without considering the new entered sub-campaign) and the value of the new considered sub-campaign (considered singularly) s.t. the daily budget over the three sub-campaigns sums to the total daily budget. Once we have filled the entire table, we have the best solution in the last row, i.e. when all the 3 sub-campaigns are considered. In figure 2.1 we can see, for each sub-campaign, the real function generating the number of clicks n(x), the function learned by GP regressor and its associated uncertainty.

2.3 Performance evaluation

In order to evaluate the performance of the implemented algorithm, we have computed the *cumulative regret* as the difference between the expected reward of the *Clairvoyant algorithm* and the expected reward of our combinatorial bandit algorithm.
In 2.2 the plot of the regret we obtain:

Figure 2.1: Functions generating the number of clicks

.

Figure 2.2: Cumulative regret of the combinatorial bandit algorithm

.

Chapter 3

Advertising: Handling Abrupt Phases

The object of this section is the design a sliding-window combinatorial bandit algorithm for optimizing the budget allocation over the three sub-campaigns in order to maximize the total number of clicks, in the case in which there are the three abrupt phases. For addressing this assignment 8 we started from the simplified case of one single phase solved in the previous section and we extended it in order to take in consideration the more general scenario of multiple phases.

3.1 What is an abrupt phase

Abrupt changes are a specific case of non stationary environment. In non stationary environments the probability distribution of the random variable associaetd with the reward of every arm can change during time. Abrupt changes are tipcally associated to new products enter in the market. In this scenario the interests of the cutomers toward the previous towards can decrease abruptly. In this particular environment the total time horizon is divided in phases and in evey phase the reward function is costant while it changes abruptly between the end of a phase and the beginning of the next one.
In Figure 3.1 we can se an example of three different curves generating the number of clicks, one for each phase, associated to a specific sub-campaign.

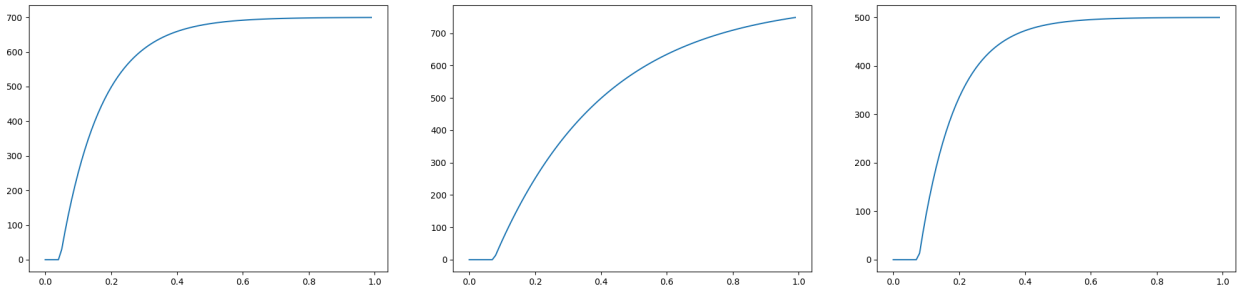


Figure 3.1: Example of different curves associated to different abrupt changes

3.2 Sliding window mechanism

For this purpose we have implemented the *AbruptBiddingEnvironment.py* class which extends the *BiddingEnvironment.py* class. This class works in a scenario of multiple abrupt phases by returning, for each sub-campaign, the reward of a given pulled arm, depending on the phase we are in.

In this case, the functions generating the number of clicks given a bid value can change dynamically, according the phase we are in. The curves differ from the previous definitions only for the dependence to the phase we are in. As follows the mathematical defintion of the three curve functions:

$$n(x) = c_M[phase] \cdot (v_M[phase] - e^{-\alpha x}) \tag{3.1}$$

We have to remark that the learner is completely transparent with respect to the actually phase we are in and in order to learn the three curves in the case of multiple abrupt phases, we have implemented the *DynamicLerner.py* class as an extension of the standard GP_Learner. It implements a *sliding window* mechanism in which we pull a new arm and add the collected rewards until the length of the window is reached. When the window is full, for each new pulled arm we get, we delete the last recent value and add the new one to the collected rewards.
Before comparing the performance of this implementation with respect to the one of the standard GP_Learner, we did a validation in order to understand what is the best length of the window to be adopted. In Figure 3.5 the regret associated to three different example of executions of the algorithm with a different windows length. In particular in the plots we can se the regrets in which the length has been set to 33, 67 and 133, i.e. rispectively $\frac{1}{6}$, $\frac{1}{3}$ and $\frac{2}{3}$ of the whole time horizon T = 200 days.

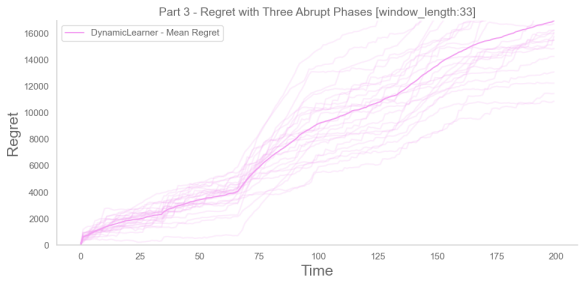


Figure 3.2: Window length = 33

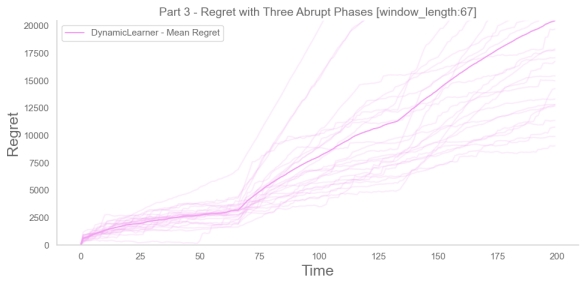


Figure 3.3: Window length = 67

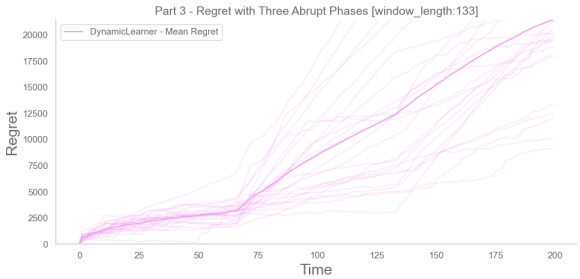


Figure 3.4: Window length = 133

Figure 3.5: **Tuning in order to find the best window length value**

As we can notice the performance is highly influenced by the choice of the window length value. Furthermore all the above results are obtained by running the algorithm over three phases characterized by the same length and so the result can differing more in less specific scenario. For this reason a new method has been designed and implemented to overcome this problem.
The final results of the performance of the sliding windows algorithm is for this postponed and it will be analyzed later, in the last section, after the description of the new algorithm we propose.

3.3 Changes Detection mechanism

For solving this problem we have implemented an innovative method which is based on the concept of the *Statistical test*. We recall that a statistical test is a procedure for deciding whether a hypothesis about a quantitative feature of a population is true or false. Then for testing a hypothesis, we draw a random sample and calculate an appropriate statistic on its items. If, in doing so, we obtain a value of the statistic that would occur rarely when the hypothesis is true, we would have reason to reject the hypothesis.

In our scenario, to detect changes, we need to compare the value of the new drawn sample with the distribution of the points belonging to the same arm in order to decide whether the hypothesis H_0,that the new drawn point belongs to that distribution, is true. Let us suppose to draw a sample whose value is x_0 and let μ σ^2 be respectively the average and the standard deviation of the the distribution of the points belonging to the pulled arm. If the point doesn't belong to that distribution the hypothesis test should reject H_0, with a *confidence interval of 99%* So we have:

$$\mathbb{P}\left(\frac{|x_0 - \mu|}{\sqrt{\sigma^2}} > h\right) = 0.01 \tag{3.2}$$

We have that the critical z-score when using a 99% confidence level are ± 2.58 standard deviations.

In the *DLChangeDetect.py* class, we have implemented such change detector. When an arm is pulled more than '**min_len**' [1] times, for each pull we run the statistical test. When the test reject H_0, it means that the point doesn't belong to that distribution and so it means that it is changed. In this case we reset the arm.
As we did for the sliding window case, also for this algorithm we performed some validation, in order to understand how to properly set the value of the *min_len*. As follows the plots of the regrets associated to three different example of executions of the algorithm by changing the value of the hyperparameter, i.e. with min_len equal to 3, 6 and 10 rispectively in Fig 3.6, 3.7, 3.8

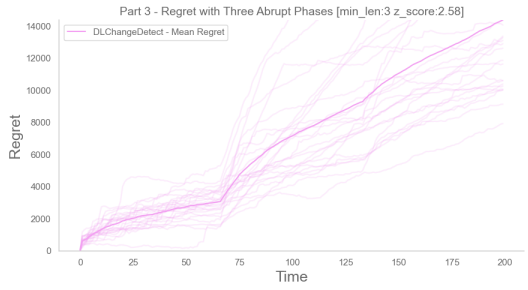


Figure 3.6: case of min_len = 3

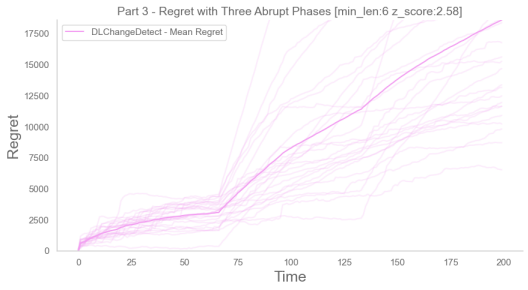


Figure 3.7: case of min_len = 6

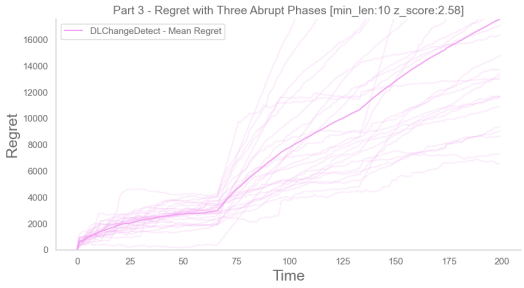


Figure 3.8: case of min_len = 10

Figure 3.9: **Selection of the best min_len hyperparameter**

Different from before, in this case the performance is not affected from the choice of the min_len parameter and for this reason, in the final study, we have chosen to run the algorithm by setting min_len equal to 10, because in this way we consider the arms with more data and so the test can be considered more reliable

3.4 Performance evaluation

This section aims to compare and describe the performance of the three described algorithms. It is important to remark that in light of the validation we have performed, the comparison are between the standard version of the GP_Learner, the Dynamic Learnear implementing the sliding window mechanism with window length equal to 33, i.e. on third of the total time horizon of 200 days and the DL_ChangeDetect learner with min_len set to 10. In particular three scenario have been tested:

[1] min_len indicates the minimum number of data we need to have in an arm in order to considere reasonable to run the test

1. Three abrupt phases, each phase with a different length
2. Three abrupt phases, all the phases having the same length
3. Four abrupt phases with different length

The first scenario is intended to analyze and compare the performance of the three different algorithms, while the the last two are more specific use cases aimed to compare the behaviour of the Dynamic Learner and the DLChangeDetect learner and to show that the considerations made before actually occur.
In Figure 3.10 the result of the first scenario is shown. As we can notice, except for the first days in which we have the first phase and so all the learner works similarly, the standard version of the GP_Learner tends to increase linearly as the number of days increases. On the other hand the two dynamic algorithms both adapts very well to the abrupt changes of the non stationary environment and tend to grows logarithmically. In the plot, we can also see that in this case the DLChange Detector quickly fits the changes while the sliding windows mechanism is slower to learn the changes. Morovere we have that the second phase was very short and this affected drastically the performance of the DynamicLearner and this is consistent to what seen before, because the behaviour of the algorithm depends on the length of the window.

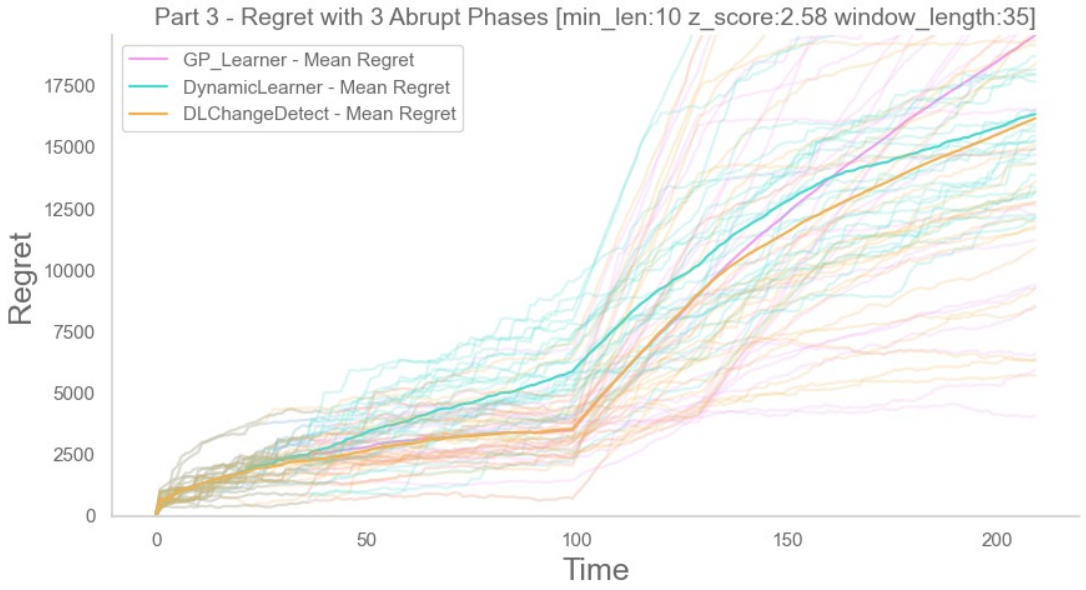
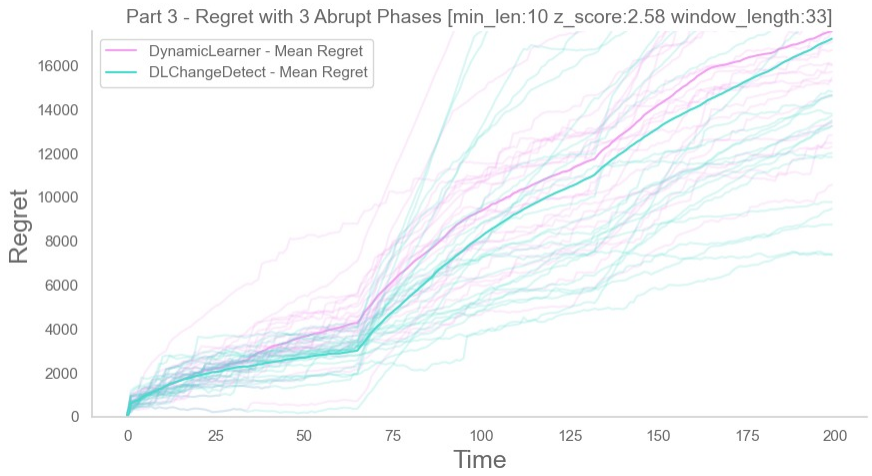


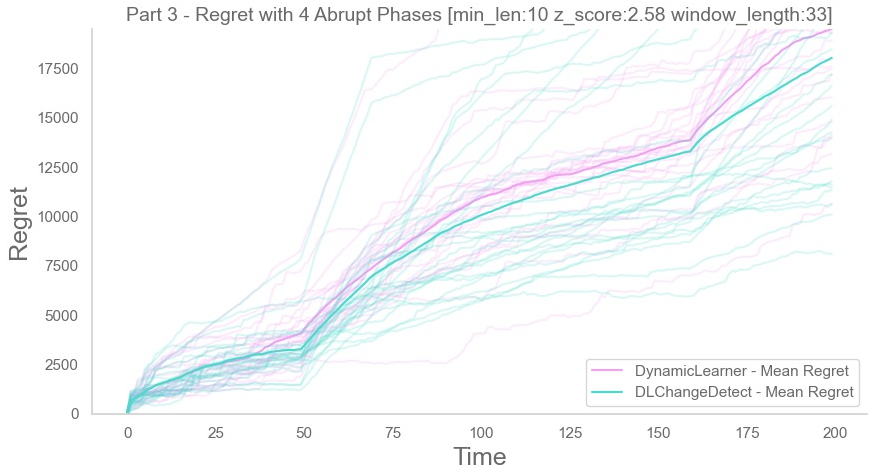
Figure 3.10: Comparison of the three implemented algothims in non stationary environment

.

Finally, as already mentioned, two more specific use cases have been considered. In the following figures we can see the comparison of the regrets of the two implemented algorithms. More precisely in figure 3.11a the case of three phases, in which each phases consists of 66 days for a total time horizon of 200 days, while in figure 3.11b the scenario of four different phases more precisely of 50, 20, 90 and 40 days.
As we can see from the two plots, the DLChange Detector always outperforms the Dynamic Learner.



(a) Performance in scenario with three phases with same length



(b) Performance in scenario with four different phases

Figure 3.11: Comparison between the regrets of the Combinatorial bandit algorithm with a sliding window mechanism and of the Change Detector

Chapter 4

Pricing: Learning the Best Price

In this chapter we are going to focus our attention on the pricing and on the optimal price, that we can propose to our customers, to maximize the total revenue.

In common scenarios, in which there is a seller that has to sell products, the retail price is fixed without exploiting information from the buyers. Sometimes, in physical markets, the customers have the possibility to negotiate and a skillful seller can maximize his revenue keeping the price as high as possible. This is a common practice, but experience is needed, to understand the limit of the buyers, and moreover it requires time, loosing the possibility to serve other clients.

In e-commerce scenarios, we can do better exploiting information from the users, from the big data and from the software automation. In particular, given the assignments 8, we are going to design an algorithm being able to learn the optimal price for our product, to maximize the total revenue.

In the following steps, we are going to:

- define the demand curves of the users;
- describe the environment setup;
- explain the algorithm that has been chosen; and
- comment the results obtained in the end of the campaign.

4.1 Demand Curves

In the economic literature is defined the concept of **demand curve**. It is a function that maps the price to a certain probability. In other words, given a price, the function returns the probability that the product will be sold. Therefore, if we plot the demand curve, we'll obtain on the x-axis the admissible prices and on the y-axis the probability (between 0 and 1).

In our simulated scenario, we defined three classes of users with the corresponding demand curves. The shapes of the curves are motivated by the intrinsic behaviors of the classes:

- *Elegant*: they are rich, thus they will buy the product with high probability whatever is the proposed price;
- *Casual*: they are students, thus they are able to buy the product only if the price is relatively low; and
- *Sports*: they are workers, but without a particular interest on the shoes, therefore the price is relatively irrelevant and the probability is low.

In figure 4.1 there are the curves designed following the above descriptions. [1]

In the same figure, the *aggregate* curve has been reported (colored in *orange*), it represents the mean between the other curves. (It will be useful in the next subsections.) Moreover, points representing the theoretical optimum are reported. The area below them is maximum w.r.t. the curves.

[1] A tool as been implemented to draw the curves manually. It requires few points and it performs polynomial regression to augment the sampling definition. It can be found in the attached code.

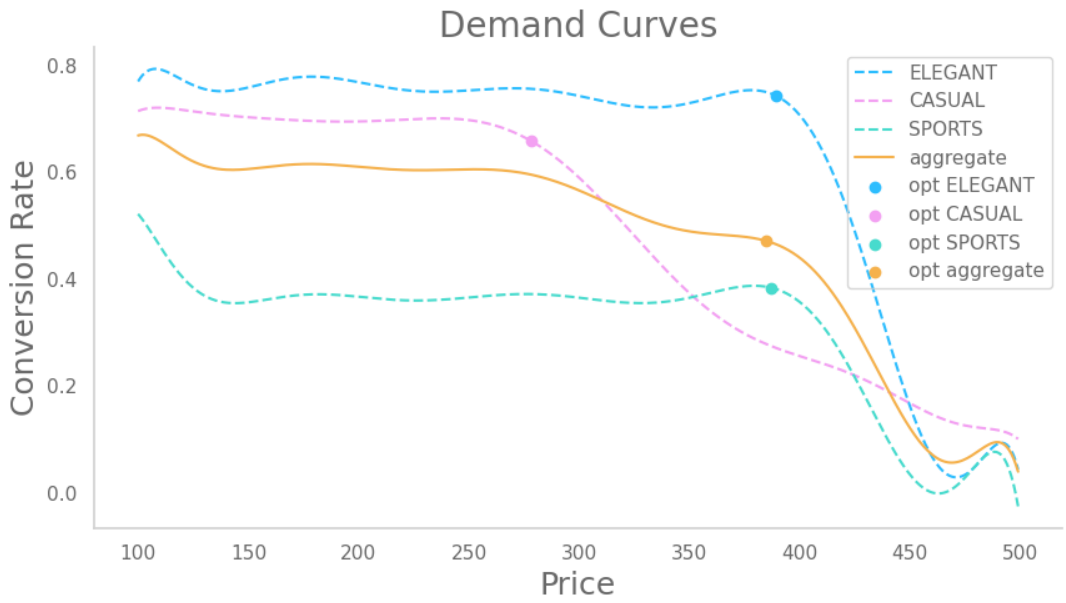


Figure 4.1: **Demand Curves**.
They are used to model the probability (*conversion rate*) that a certain class, of customers, will buy or not the product w.r.t. a proposed price.
The dashed lines represent the demand curves of the corresponding classes: *blue* for the *Elegant* class; *pink* for the *Casual* class; and *light green* for the *Sports* class.
The *orange* curve is the mean between the others and it is used to study the *aggregate* model.
The points are displayed to show the optimal price w.r.t. the conversion rate: the area below them is maximum.

4.2 Environment

In our implementation, the environment is a black box from the point of view of the learning algorithm. It is used to iterate over the days, for the entire duration of the campaign. It schedules the time and the number of users reaching our theoretic website.

Every user is randomly taken from the three classes and the class is not communicated to the learner. The environment, knowing the original class of the users, computes the optimal revenue and returns the effective reward obtained by the learner. How the learner will handle these information will be explained in the next subsections.

4.3 Learner Algorithm Selection

This context of application can be solved in different ways, but the most common choice is to use a *multi-armed bandit (MAB)* algorithm.

We mainly tested two algorithms:

- **Thompson Sampling (TS)** algorithm; and
- **Thompson Sampling with Sliding Window (SWTS)** algorithm.

We tested both the algorithms during the implementation phase, but since the SWTS is more sensible to the window size, and finding a good one requires a lot of testing, we kept the TS as final choice.
This decision has been motivated by the fact that there aren't abrupt phases, thus there is no impelling reason to "forget the past" as done by SWTS.

An additional note must be done for the pulling phase of the TS algorithm. The standard implementation of the TS pulls the arm in this way:

$$a_t \leftarrow \underset{a \in A}{\operatorname{argmax}}\{\tilde{\theta}_a\} \tag{4.1}$$

Where A is the set of arms; $\tilde{\theta}_a$ is the sampled random value from the beta distribution for arm a.
This implementation maximize the demand, but since we are interested in the maximization of the total revenue, we pull the arms as below:

$$a_t \leftarrow \underset{a \in A}{\operatorname{argmax}}\{\tilde{\theta}_a \cdot price_a\} \tag{4.2}$$

Where $price_a$ is the price targeted by the arm a.
This allow us to maximize the area under the demand curve.

4.4 Experiments

Summarizing, we have to learn the best price to propose, to the users, to maximize the total revenue.
We have discussed about the environment, that generates incoming users and allows us to iterate over the days of the campaign, and finally, we presented the learner adopted.
At this point we can explain how the tests have been initialized.

The experiments has been configured in two ways:

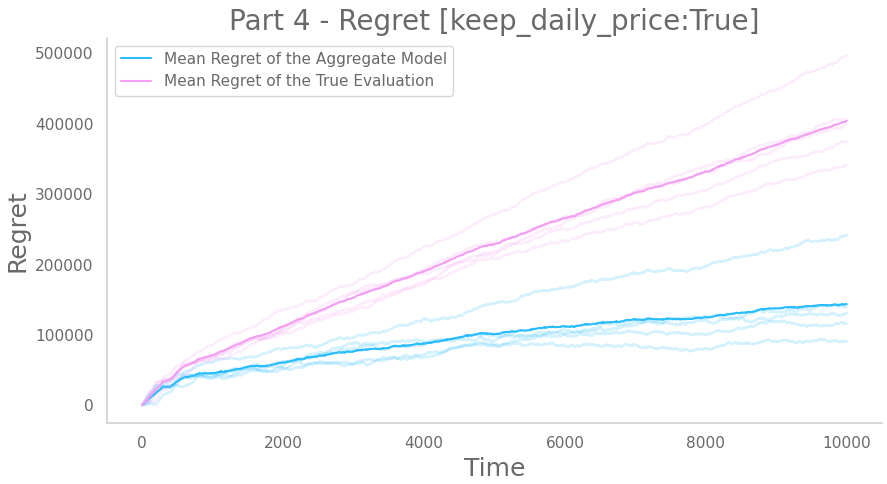
- keeping a fixed daily price: we pulled an arm only for the first user of the day; for the other users we proposed always the same price. This means that for the entire day we pull and update always the same arm of the TS;
- one price for each user: maybe, this is an unpractical option in a real world scenario, but it allows to obtain a total revenue outperforming the previous configuration.

In the end of the tests, the figure 4.2 reports the obtained regrets. The sub figure *(a)* for the first configuration and the sub figure *(b)* for the second.

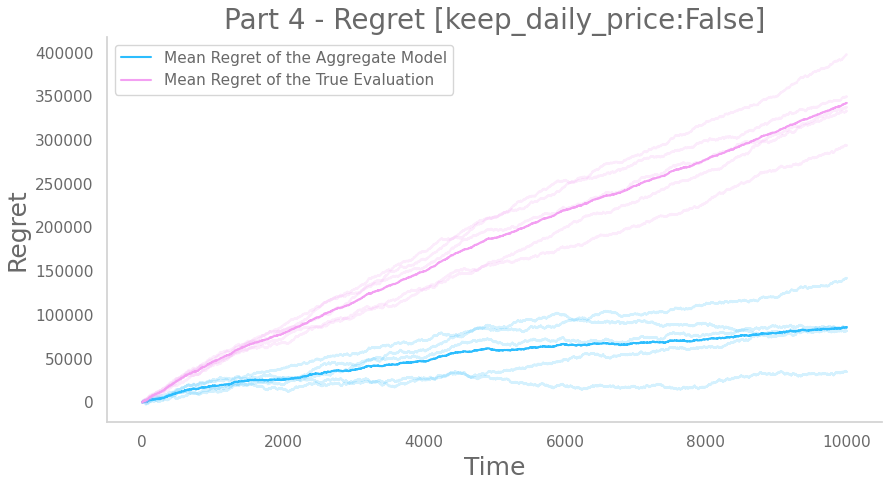
We plotted two regrets computed with two different optimal revenues. The regret called **Mean Regret of the Aggregate Model** has been computed using, as optimal, the optimal point taken from the aggregate model. This is an optimistic regret because it doesn't take into consideration the information about the specific class of the users. In other words, the learner and the clairvoyant algorithm don't know the class of the customers.

The learner is learning the aggregate model, thus this comparison seams fair, but since the clairvoyant knows the original class of the users, we introduced the so called **Mean Regret of the True Evaluation**. Obviously, from the point of view of the learner, this is a pessimistic computation since it is unable to infer more information from the input data.

Anyway, these two ways, to compute the regrets, will be useful in the next chapter, since the learner will be able to infer more information from the data, creating contexts to exploit a finer profiling. You will notice how the two regrets will be pushed down and the regret on the True Evaluation will be more similar to the regret of the Aggregate Model obtained here, proving that a finer profiling improves drastically the performances.



(a) Regret obtained proposing to the customers the same price for the entire day.



(b) Regret obtained proposing to the customers different prices for each visit. This method achieves better performances since the learner collects more data during the campaign.

Figure 4.2: Comparison between the regrets obtained during the campaign.
The **Mean Regret of the Aggregate Model** (colored in *blue*) is the regret computed keeping, as optimal, the area under the optimal point of the aggregate curve 4.1. In other words, both the learner and the clairvoyant don't know the class of the customers. This is an optimistic measure of the regret.
The **Mean Regret of the True Evaluation** (colored in *pink*) is the regret obtained computing the optimal value exploiting the original class of the customers.

Chapter 5

Pricing: Context Generation

In this chapter we want to further expand what we have seen in the previous one. In the general case, since we have access to many information about our clients, we can not only identify different classes of users, with different buying powers, but also understand to which class a new user belongs, based on the information we have about him (given, for example, by cookies). The idea is to exploit this knowledge in order to charge different prices to each class and so maximize our revenue. To do so, we will use the idea of the **context**, which will be explained in the next section. We will then explain the differences in the environment used in this chapter, tackle the approaches we followed to create the algorithm and analyze the final results.

5.1 What is a Context?

As said before, in this scenarios we have a lot of information about the users that visit our website, so we can group them together in classes. Usually a class is identified by a single value (or subsets of values) for each feature. The issue is that the number of classes grows exponentially with the number of features, making unfeasible the problem of defining and optimizing a single price for each class when we have many features for each user (which is the most common scenario in real cases). A context is a partition of the feature space according to some values of the features (for example a male with age between 18 and 24 is a context), grouping together various classes. By using contexts instead of classes, we are able to drastically reduce the number prices we have to decide, so the problem can be tackled and solved in a reasonable time.

5.2 Environment, Learner and Demand Curves

The environment is fairly similar to the one used in Chapter 4, the only difference is that it keeps track of how many days have passed since the collected data has been analyzed, so that we know when we have enough to consider generating the new contexts. This is possible because we know how many users will visit our website every day, so we know exactly how much data we have after each day. We used the same demand curves as in Chapter 4, on the other hand, since we want to learn different prices now, we don't use a single learner anymore, but we create one with each new context.

5.3 Campaign Scheduler

The class Campaign Scheduler acts as the main interface for our algorithm. It stores the contexts we are currently using for the week and the information collected by our website so far in a structure indexed using the classes we have chosen. These information are updated after each user visits the website and will then be used in the Context Generator class to decide whether to create new contexts or not.

- *Total Reward*: The sum of all the rewards we have obtained so far;
- *Number of purchases*: The number of products we have actually sold;
- *Number of users*: The number of users who have visited our website;
- *Rewards*: An array that contains all the rewards collected so far.

5.4 Our Approach to Context generation

When we first approached the problem, we used a algorithm similar to that presented in class, with a greedy approach. We start with the aggregate model, so we have only one price for all the classes, and we use it to collect data for the first week. At the end of the week, we use this data to evaluate the features and decide if we want to split and generate new contexts. For each feature we have yet to use to split, we compute the lower bound of the probability of the context to occur and of the expected reward. This is because when we create a new context, we have to learn a different model, which will surely have more uncertainty with respect to the one used so far. The lower bound is a measure of how much we are guaranteed to gain, the reward we collect in the end might be higher.

For the lower bound of the probability we used the Hoeffding bound since the probability has limited values.

$$x - \sqrt{-\frac{\log_n d}{2n}} \tag{5.1}$$

where:

- n : is the number of samples.
- x : is the mean of the rewards.
- d : is a value that defines how much confidence we want.

For the reward we had to use a different formula, since its values are unlimited.

$$x - t\sqrt{\frac{sq}{n}} \tag{5.2}$$

where:

- n : is the number of samples.
- x and sq : are the mean and variance of the real rewards.
- t : is a value that defines how much confidence we want.

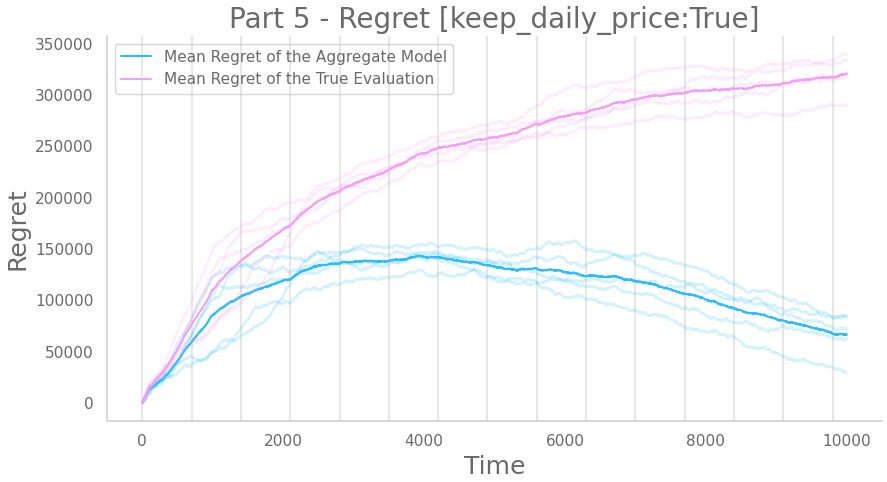
After evaluating the expected reward relative to each feature, we select the one with the highest value and compare it with the expected reward of the current model. If it is larger or equal we create the new contexts, if not we keep the previous model and start a new week. This process is iterated week after week, without a termination condition, because we are continuously collecting information which might change how our contexts are organized. In the general case this algorithm does not guarantee optimality and is used to solve scenarios in which we have many features with possibly many values thanks to its performances. On the other hand it gives us some insight on which features might be the most important in the scenario.

However our problem was quite different from the general one, so we chose a different approach: evaluating every week the entire set of possible partitions. Since we have only two features with binary values, this process is not computationally heavy and we can use this solution with reasonable time performances. We always start with the aggregate model to collect data, but when the time to split arrives, we evaluate all possible partitions of our feature space, using the above formulas. This way we can either obtain the aggregate model, two or three contexts, by not splitting, splitting according to only one or both features (we would obtain four contexts, but we only have three classes).

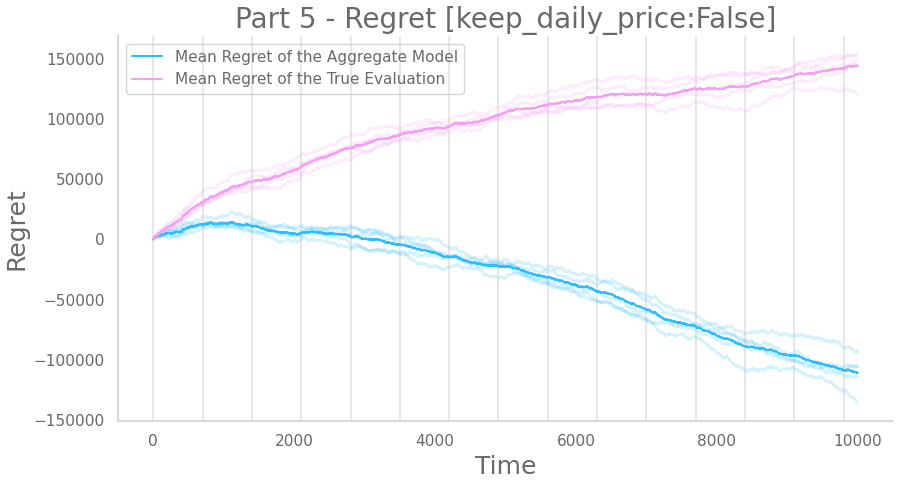
5.5 Results

As in Chapter 4, we configured the experiment to both keep the same price for the day and not to and this is the reason of the two graphs. Moreover we used the same two optimal models, the Aggregate Model, which does not know the classes of the users, and the True Evaluation, which does and chooses a single price for each of them. This was done to compare the results of the algorithms we have built.

As we can see in figure 5.1, the algorithm largely outperforms the Aggregate Model, because it knows the classes of the users and can choose a price for each of them. In the first weeks we observe a small growth before the decrease, this is because the algorithm starts from the aggregate model and is still learning the correct prices. We can also see that the initial growth lasts longer when we keep a daily price, because by fixing the price it takes longer to learn the curve. This comparison was mainly done to show the improvements with respect to the algorithm in Chapter 4. On the other hand, the True Evaluation model performs better than the algorithm in the first weeks, but as time goes on the difference becomes smaller and smaller, as the algorithm learns the best prices.



(a) Regret obtained proposing to the customers the same price for the entire day.



(b) Regret obtained proposing to the customers different prices for each visit.

Figure 5.1: Comparison between the regrets obtained during the campaign.
The **Mean Regret of the Aggregate Model** (colored in *blue*) is the regret computed keeping, as optimal, the area under the optimal point of the aggregate curve 4.1.
The **Mean Regret of the True Evaluation** (colored in *pink*) is the regret obtained computing the optimal value exploiting the original class of the customers.
The vertical lines are positioned every seven days, to underline when the new weekly contexts are generated.

Chapter 6

Adverting and Pricing Integration

Reached this point the goal is to attract the most valuable users, that is allocate the budget during Advertising so that it is focused on attracting users that will maximize the total reward; this expectation on the reward is modeled by the pricing algorithms presented before. In this section we will present how we have modeled the intregation between Pricing and Advertising section, which are the algorithm proposed and an analysis of the final results.

6.1 Proposed Integration

The idea behind our integration is to have, for each Subcampaign, the possibility to manage separately the Advertising and the Pricing in order to reuse all the algorithms presented in the previous sections and, on top of it, build a Budget Allocator that is able to collect all the information and calculate the best allocation of the budget. The complete flow chart can be found at Figure 6.1, in the following sections we will discuss it step by step.

6.1.1 Budget Allocator

Every day, this component is the one that, collected information from the past, calculates the best budget allocation. Initially, when we have no historical information, it is built to output a balanced allocation among all the Subcampaigns. After the initialization phase, evey new day, it collects for every subcampaign j two values:

1. $n_j(.)$: Learned distribution of clicks over budget allocation for subcampaign$_j$. This function comes from the regression generated by the GPTS learner in the **Advertising** section.
2. v_j : Learned value, in terms of expected reward for subcampaign$_j$. This value comes from the evaluation of the collected reward during the last days in the **Pricing** section.

At this point we are able to define the Knapsack Optimization Problem as:

$$\max_{y_{j,t}} \sum_{j=1}^{N} v_j n_j(y_{j,t})$$

$$\sum_{j=1}^{N} y_{j,t} \leq \bar{y}_t$$

The complete legend can be found at Figure 6.1

6.1.2 Advertising

For each subcampaign$_j$, given the budget allocation $y_{j,t}$, it is possible to collect from the environment the real number of clicks $N_j(y_{j,t})$ generated by this allocation and update, as we have done in Chapter 2 and 3, the GPTS learner. At the end it is also returned the distribution $n_j(.)$ learned by the Gaussian Process.

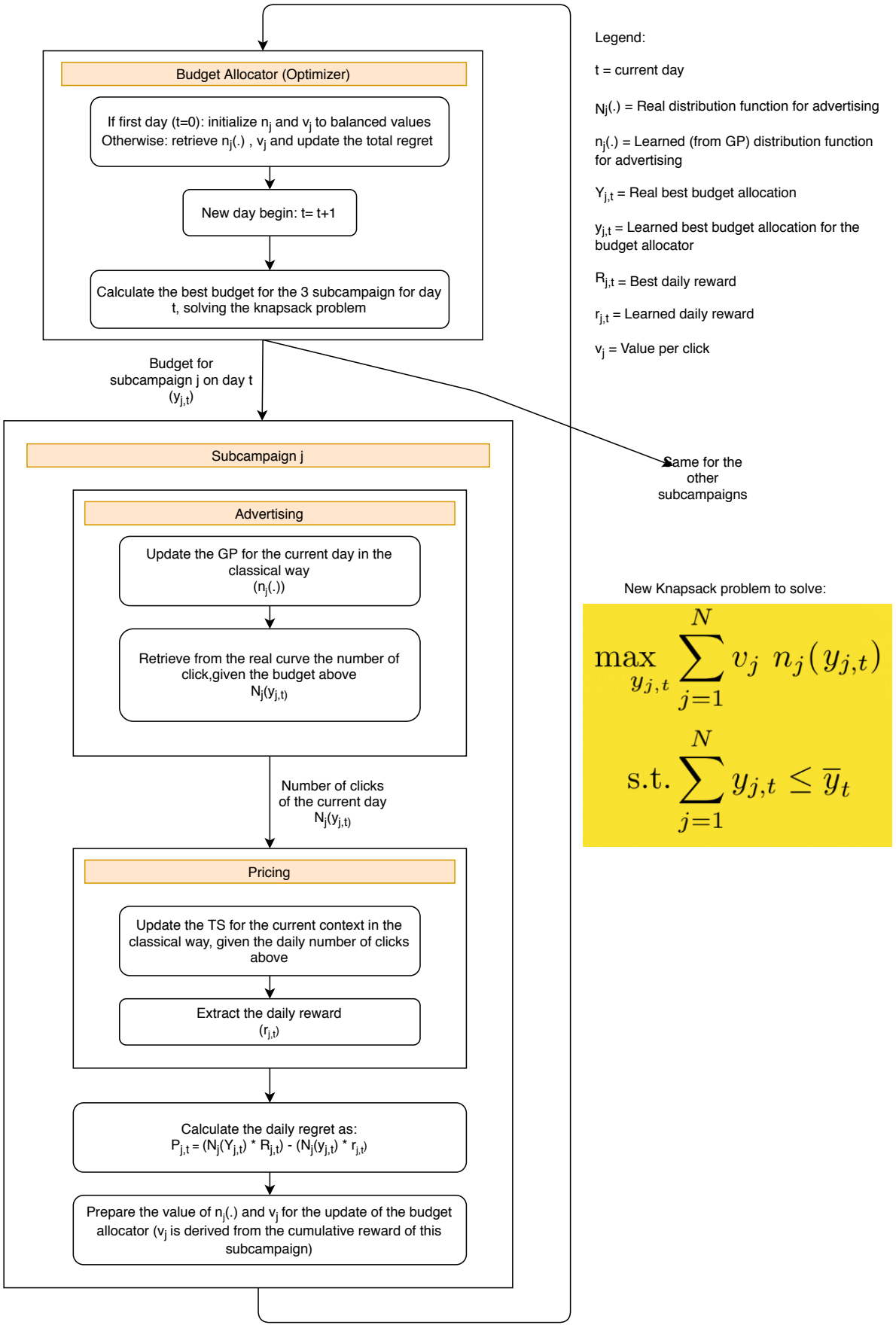


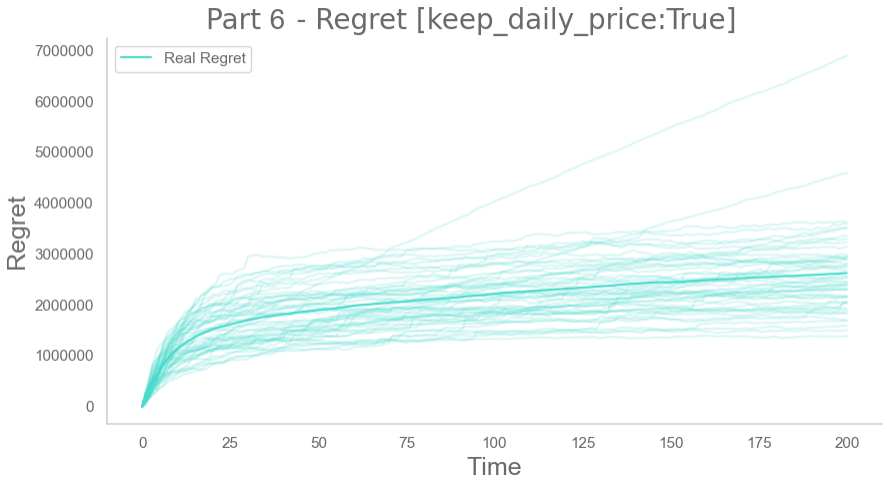
Figure 6.1: Flow Chart Diagram of part 6

6.1.3 Pricing

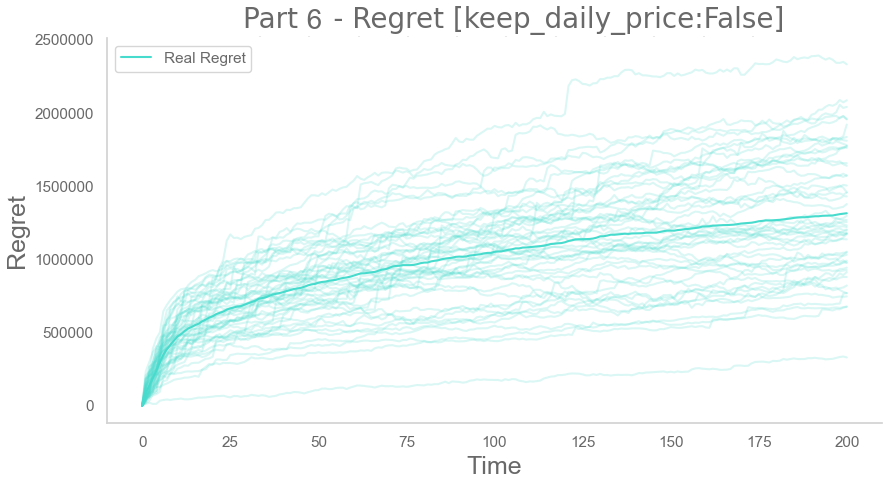
For each subcampaign$_j$, given the real number of clicks $N_j(y_{j,t})$ generated by the **Advertising**, it is possible to collect the real revenue $r_{j,t}$ for the current day and update, as we have done in Chapter 4, the TS learner. At the end it also returns the value v_j of this subcapmpaign, by averaging the collected revenue of the last N days. The reason of this choice is that we want to balance the trade-off between exploration (few days) and exploitation (more days). After validation runs we have chosen N=5 days.

6.2 Performance Evaluation

At the end of each day, given $r_{j,t}$ and $R_{j,t}$, the optimal daily reward, the regret of the current day for every subcampaign is calculated. As in Chapter 4 and 5, we configured the experiment to both keep the same price for the day and not to and this is the reason of the two graphs. It is possible to see that, in agreement of what presented before, the regret in which is proposed different prices for each visit is smaller than the opposite case. 6.2.
We have also evaluated the final loss, calculated as the fraction between the total reward divided by the total regret and it is around 2.5% (while the loss of the version with keeping price fixed is 5.6%).
The experimental results come from the evaluation of 50 different experiments on 200 days.



(a) Regret obtained proposing to the customers the same price for the entire day.



(b) Regret obtained proposing to the customers different prices for each visit.

Figure 6.2: Comparison between the regrets obtained during the campaign.

Chapter 7

Advertising and Pricing: Fixed Prices

Do the same of Step 6 under the constraint that the seller charges a unique price to all the classes of users. Suggestion: for every possible price, fix this price and repeat the algorithm used in Step 6. Plot the cumulative regret when the algorithm learns both the conversion rate curves and the performance of the advertising subcampaigns.

Chapter 8

Assignments

Part 2:
Design a combinatorial bandit algorithm to optimize the budget allocation over the three subcampaigns to maximize the total number of clicks when, for simplicity, there is only one phase. Plot the cumulative regret.

Part 3:
Design a sliding-window combinatorial bandit algorithm for the case, instead, in which there are the three phases aforementioned. Plot the cumulative regret and compare it with the cumulative regret that a non-sliding-window algorithm would obtain.

Part 4:
Design a learning algorithm for pricing when the users that will buy the product are those that have clicked on the ads. Assume that the allocation of the budget over the three subcampaigns is fixed and there is only one phase (make this assumption also in the next steps). Plot the cumulative regret.

Part 5:
Design and run a context generation algorithm for the pricing when the budget allocated to each single subcampaign is fixed. At the end of every week, use the collected data to generate contexts and then use these contexts for the following week. Plot the cumulative regret as time increases. In the next steps, do not use the generated contexts, but use all the data together.

Part 6:
Design an optimization algorithm combining the allocation of budget and the pricing when the seller a priori knows that every subcampaign is associated with a different context and charges a different price for every context. Suggestion: the value per click to use in the knapsack-like problem depends on the pricing, that depends on the number of users of a specific class interested in buying the product. Notice that the two problems, namely, pricing and advertising, can be decomposed since each subcampaign targets a single class of users, thus allowing the computation of the value per click of a campaign only on the basis of the number of clicks generated by that subcampaign. Plot the cumulative regret when the algorithm learns both the conversion rate curves and the performance of the advertising subcampaigns.

Part 7:
Do the same of Step 6 under the constraint that the seller charges a unique price to all the classes of users. Suggestion: for every possible price, fix this price and repeat the algorithm used in Step 6. Plot the cumulative regret when the algorithm learns both the conversion rate curves and the performance of the advertising subcampaigns.