Pricing & Advertising

Gabriele Gavarini

December 14, 2020

Contents

Chapter 1

Data Model

In the following chapter we present the data used for testing our algorithms. The full dataset is script-generated and can be found in the file "Scenario/User_pricing.csv".

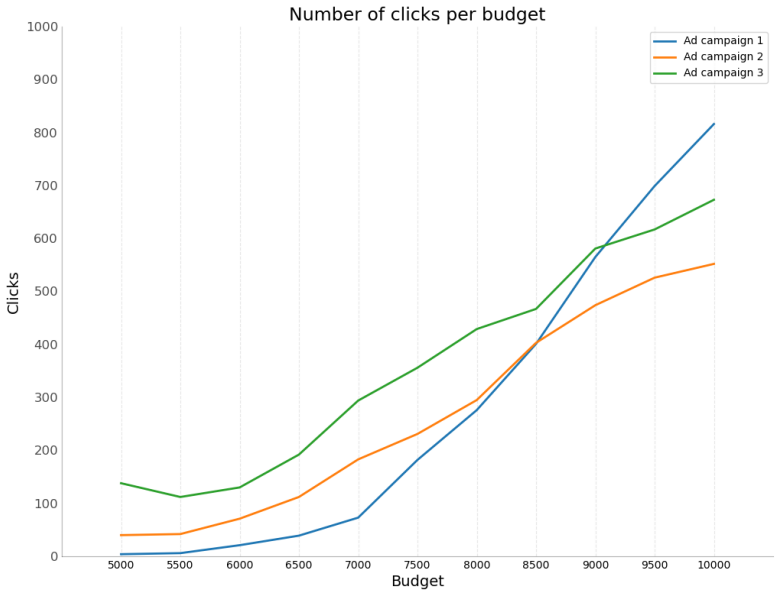

Figure 1.1: Click per budget, assuming no difference in phases.

1.1 Model description

This script-generated model contains a set of statistic about the efficacy of the advertising campaigns, other than information related to the pricing of a given product. No further description of the item and of the campaigns are provided as they would be meaningless to the functioning of our algorithm.

1.1.1 Advertising

For what concerns the advertisement of the product, we model 3 different campaigns, each composed of 3 different phases. For every campaign, we can define a function that maps a budget spent on that campaign to a number of users that will click on the ad of that campaign. The phases are used to model 3 abrupt changes to this function.

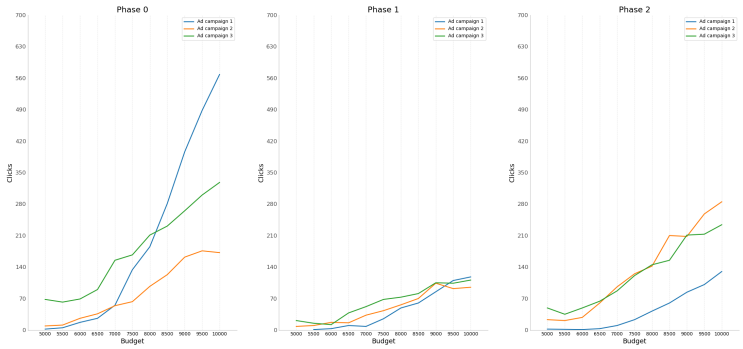

Figure 1.2: Clicks per budget, split in 3 different phases

In **figure 1.1**, we can see the generic efficacy of the 3 different policies, while in **figure 1.2** we can see the different behaviors of the campaigns during the 3 phases.
Ad campaign 1 is a campaign targeted to females that are older than 30. We can see that this campaign is very effective in the first phase, but loses all it potential in the following phases.
Ad campaign 2 is a campaign targeted to females that are younger than 30. While initially this is the worst campaign in term of clicks, it is the most cost effective in the last phase.
Ad campaign 3 is a campaign targeted to males. It has a similar behavior to the second campaign.

1.1.2 Pricing

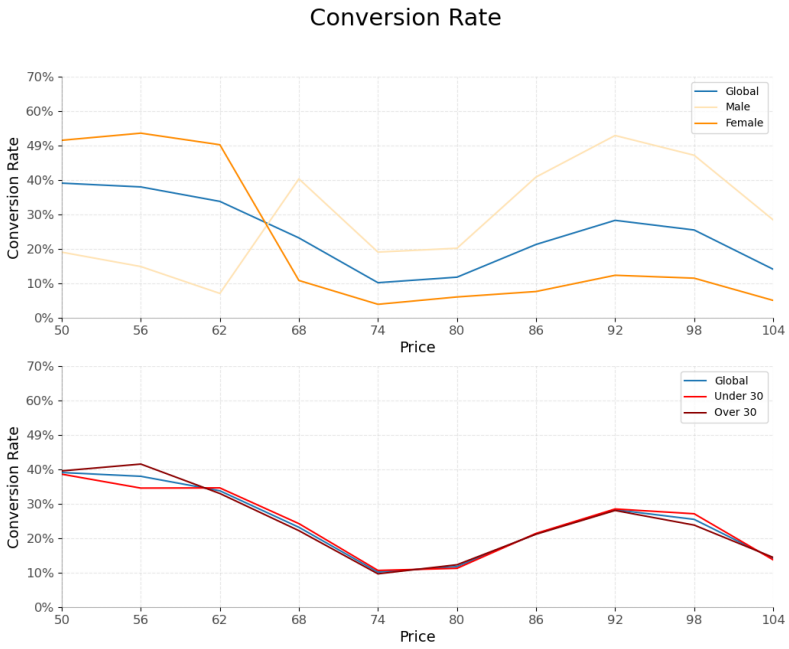

Figure 1.3: Conversion rate for different class of users

Here we model how an user reacts to the price of the product. We will analyze how many user will buy the product based on the price it has been proposed to them, the so called **conversion rate**.

As shown in **figure 1.3**, while the conversion rate curve isn't affected by the age of the costumers, it's strongly dependent on their sex. It follows that an ideal strategy should be to propose different prices according on the consumer's sex. Since in **chapter 5** we will analyze how to select the best way to partition the users, we propose one of the worse situation possible: we group users based on their age but not on their sex.

1.2 Dataset structure

Every entry in the dataset represent a user that has clicked on ad. Initially, we had two distinct tables, one containing the information about the ad, and another containing the information about the costumer. For the sake of simplicity, we have merged those information in a single dataframe.

The column present in the dataset are the followings:

- `Phase`: ad-related, represents the phase of the advertising campaign in which the ad was clicked.
- `Ad_campaign`: ad-related, represent the campaign to which the clicked ad belongs.
- `Budget`: ad-related, represent the amount spent on the campaign.
- `Price`: the price of the advertised product.
- `Converted`: whether or not the user has bought the product after clicking on the ad.
- `Under_30`: user-related, if the user is under or over 30 years old.
- `Sex`: user-related, if the user is male or female

Chapter 2

Budget Allocation - Single phase

In this chapter we want to allocate a budget to 3 advertising campaigns in order to maximize the number of clicks on the ads. The daily budget is constrained.

2.1 Design philosophy

The **environment** we model in this chapter is composed by **3 Scenarios**: each of them represents an advertising campaign. It is possible to allocate a certain daily budget to every campaign. The performance of a scenario is measured by how many users see one of its ad and click on it. However, the relationship between the amount spent on the campaign and the resulting number of clicks is unknown. Furthermore, the amount of money we can spend each day, across all the campaigns is **constrained** by a budget cap.

Our objective is to maximize the total number of clicks and allocate to every campaign a budget such that the total amount spent on the advertising campaigns is less than the budget cap. As such, the actual problem we are studying can be expressed as:

$$\begin{cases} \max_{y_j} \sum_{j=1}^{N} n_j(y_j) \\ \sum_{j=1}^{N} y_j < Y \end{cases} \tag{2.1}$$

Where j is a subcampaign, y_j is the budget allocated to it, n_j the number of clicks and Y is the budget cap.

We split the problem in two, requiring:

- A **Gaussian Process Thompson Sampler** for every campaign j, to estimate the corresponding number of clicks n_j;

- A **knapsack-like** algorithm to decide how to allocate the budget to the advertising campaigns based on the results of the Thompson Samplers.

As such, for every timestep, we will execute the following loop:

1. Choose the arm to be pulled with the knapsack-like algorithm based on the models estimated by the samplers in the previous time instances. The algorithm is in charge of not going over budget;
2. For every subcampaign, sample the real-world function by playing the arm chosen at the previous step;
3. For every subcampaign, update the model of the Thompson sampler based on the result of the arm played in the previous step.

2.1.1 Classes

`Scenario`: represents an advertising campaign.
It provides the method:

- `round(daily_budget_index)` to obtain a response to the campaign for a given allocated budget `daily_budget_index`.

`GPTS_Learner`: is the Gaussian Process Thompson Learner associated to an advertising campaign and is responsible for the estimation of the number of clicks.
It provides the methods:

- `sample_values()` to obtain the value of the modeled number of clicks function for every possible allocated budget.
- `update(arm_index, reward)`, `update_model()` and `update_observations(arm_index, reward)` to update the model based on the result of the sampled arm.

`Optimizer`: is in charge of executing the knapsack-like algorithm.
It provides the method:

- `optimize(sampled_values)`, that select which arm to play for every subcampaign based on the `sampled_values` and while constrained by the daily budget.

2.2 Execution

Using the data from the user pricing model, we can model a number of clicks curve from each campaign, as shown in the first chapter. In particular, we can

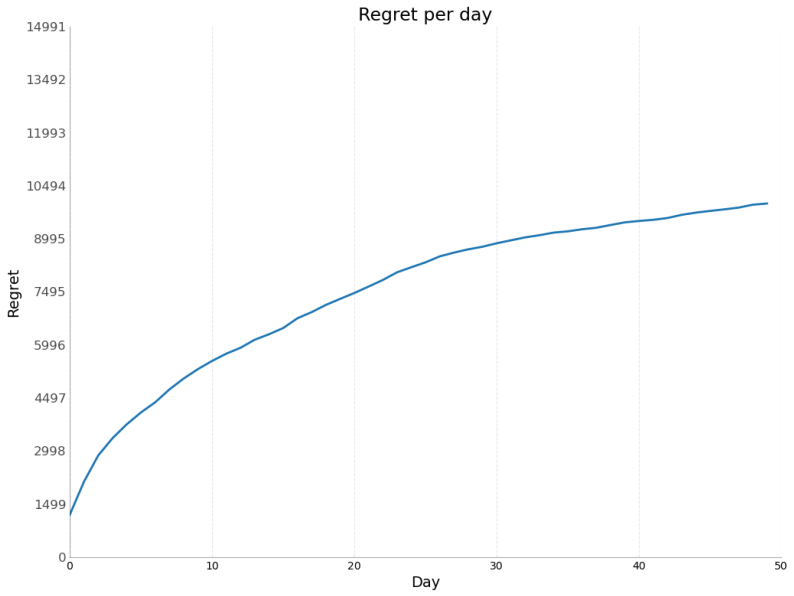

Figure 2.1: Regret cumulative sum per iteration

assign to every campaign a budget from 5'000$ to 10'000$ (sampled every 500 dollars). The daily budget cap is set to 27'000$.

Every day, for 50 days, we execute the loop described in **Section 2.1**, computing the daily **optimal solution**. Furthermore, we execute the knapsack algorithm with the actual known value of the number of clicks; we refer to this as the **ideal solution**. This allows us to compute the **daily regret** by subtracting to the ideal solution the optimal solution. Notice that, as we sample the model with noise to simulate a realistic setting, it may happen for the optimal solution to have an higher number of clicks than the ideal solution.

The experiment is repeated 10 times, to obtain a less biased computation of the regret.

As we can see in **figure 2.1**, the regret increases rapidly in the first few days and then tends to stabilize. The small increase in the later days is caused by a noisy sampling. The stabilization of the curve indicates that our algorithm has found an optimal solution; the solution found is to allocate 10'000$ to the first campaign, 9'000$ to the second one and 8'000$ to the third one. This result in 1700 daily clicks. The optimal solution found coherent with the ideal solution.

Chapter 3

Budget Allocation - Three Phases

In this chapter we want to allocate a budget to 3 advertising campaigns in order to maximize the number of clicks on the ads. The daily budget is constrained. Differently from the previous chapter, here we model an evolution in the efficacy of the campaigns.

3.1 Design philosophy

Similarly to the previous case, the environment we model is composed by **3 Scenarios**: each one of theme represents an advertising campaign and present 3 distinct phases. Just like before, it is possible to allocate a budget to every subcampaign and in return we will have a number of clicks, obtaining thus a function that is, in this case, time dependent.

Our objective is to provide a flexible way to select the optimal budget allocation such that, whenever this function changes, our algorithm adapt and select a new optimal allocation. In this case, the problem is similar to the one of chapter 2:

$$\begin{cases} \max_{y_j} \sum_{j=1}^{N} n_j(y_j, t) \\ \sum_{j=1}^{N} y_j < Y \end{cases} \tag{3.1}$$

Where j is a subcampaign, y_j is the budget allocated to it, n_j the number of clicks, t is the current timestep and Y is the budget cap. Notice how, differently from **equation 2.1**, the number of clicks is dependent on the current timestep.

As in chapter 2, we split the problem in two, requiring:

- A **Sliding-windows Gaussian Process Thompson Sampler** for every campaign j, to estimate the corresponding number of clicks n_j based on the past `window_length` results;
- A **knapsack-like** algorithm to decide how to allocate the budget to the advertising campaign based on the results of the Thompson Sampler.

The loop is the same as the one for the previous chapter, that is, choose the arms with the knapsack-like algorithm, play the chosen arms and update the models. However, in this case not all the previously sampled data is used, but only the most recent `window_length` records.

3.1.1 Classes

`ThreePhasesScenario`: represents an evolving advertising campaign. The phase changes according to an internal clock.
It provides the method:

- `round(daily_budget_index)` to obtain a response to the campaign for a given allocated budget `daily_budget_index`.
- `advance_time()` and `advance_phase()` to, respectively, advance the internal counter and, eventually, the phase.

`SW_GPTS_Learner`: is the Gaussian Process Thompson Learner associated to an advertising campaign and is responsible for the estimation of the number of clicks. In this case we manually set a `window_length` to constraint the model to the last results.
It provides the methods:

- `sample_values()` to obtain the value of the modeled number of clicks function for every possible allocated budget using only the most recent `window_length` samples.
- `update(arm_index, reward)`, `update_model()` and `update_observations(arm_index, reward)` to update the model based on the result of the sampled arm.

`Optimizer`: is the same of chapter 2

3.2 Execution

Using the data from the user pricing model, we can model a dynamic number of clicks curve from each campaign, as shown in the first chapter. In particular, we can assign to every campaign a budget from 5'000$ to 10'000$ (sampled every 500 dollars). The daily budget cap is set to 27'000$. There are 3 phases for each campaign, each lasting 100 days.

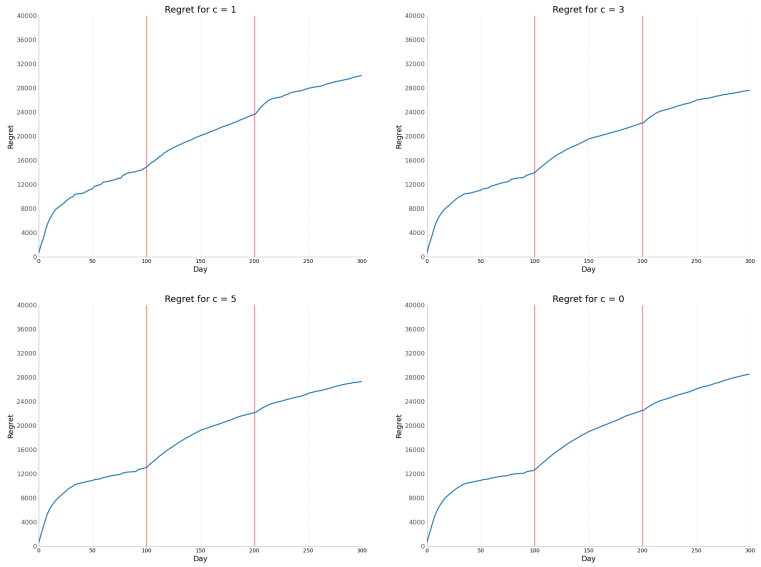

Figure 3.1: Regret cumulative sum per iteration, expressed in number of missed clicks.

Every day, for 300 days, we execute the loop described in **Section 3.1**, computing the daily **optimal solution**. In this case, the optimal value is not based on the whole history of results, but only on the `windows_length` most recent one. This value is computed as:

$$\texttt{windows\_length} = c * \sqrt{T}$$

Where c is a windows coefficient we can tune and T is the time horizon, 300 in this case.

The experiment is repeated 5 times, to obtain a less biased computation of the regret.

We execute the knapsack algorithm with the actual known value of the number of clicks; we refer to the solution found as the **ideal solution**. Notice that the ideal solution changes whenever we enter a new phase and, as such, we need to compute it again every 100 days. Those computations allow us to compute the **daily regret** by subtracting to the ideal solution the optimal solution. Notice that, as we sample the model with noise to simulate a realistic setting, it may happen for the optimal solution to have an higher number of

clicks than the ideal solution.

Figure 3.1 shows the regret for different values of the of the window coefficient c (and, therefore of the window length). We can observe how a different sizing impact how our algorithm adapts to new changes. For example, having $c = 1$ (`windows_length`≈ 17) implies that the algorithm reacts quickly to an abrupt change of the function. However, the solution found is never a good approximation, as it lacks in number of samples.

In the bottom right corner of **figure 3.1** we can observe the behavior of the algorithm when there is no window (*i.e.* $c = 0$). As expected, in the first phase, the regret behaves better than the cases where there is a window, no matter the size, losing less than 12'000 clicks with regard to the ideal solution. By comparison, the solution where we have $c = 2$ (`windows_length`≈ 34) –in the top-left corner of **figure 3.1**– shows a loss of over 14'000 clicks.

Notice that, at the end of the 3 phases, the algorithm employing $c = 2$ obtains a cumulative loss of approximately 27'000 clicks, while the algorithm with no windows obtain more than 28'000 losses. This means that the solution with no windows is almost 5% worse than the solution with a window of approximately 34 days.

This highlights the strength of a windowed model: while its performance in a static environment are limited by the size of the window, it works well in a dynamic world, where it can better adapt to an abrupt change in the click/budget function.

As a final remark, notice how crucial the choice of the right window size is. In our problem, choosing a small windows improves the performances (compared to the case with no window). However, if we choose a windows that is too big, the performance is worst.

Chapter 4

Pricing

What we want to achieve in this chapter is to identify the optimal price in order to maximize the user conversion rate (*i.e. how many people will buy the product*)

4.1 Design philosophy

In this chapter we are simulating a static **environment**: a user sees an ad, clicks on it, and will decide whether to buy the advertised product or not based on its price. We refer to the percentage of users that buy the product at a given price as the **conversion rate** of that price. Our objective is to find which price will result in the best conversion rate possible.
In order to achieve this, we want to model the conversion rate for a fixed set of prices, thus modeling a **conversion rate curve** for the advertised object.
However, being interested only in the maximization of such function, we simply need to have a good approximation of the curve around the maximum. This can be easily achieved with a **Thompson Sampler**, executing at each timestep the loop:

- Pull an arm from the Thompson Sampler according to its internal beta distribution;
- Play the pulled arm in the modeled Scenario;
- Update the Thompson Sampler's probability distribution with the newly obtained data.

Given this, we are going to implement two different items: a model of our Scenario, from which we can sample the conversion rate for a given price, and a Thompson Sampler, that we will use to learn the conversion rate.

4.1.1 Classes

`Learner`: base class of `TS_Learner` it represents a simple learner that records arms payed and related rewards. It provides the method:

– `update_observations(pulled_arm, reward)`: save the last arm played `pulled_arm` and the related `reward`.

`TS_Learner`: extension of `Learner`, is the class that implements the Thompson Sampler. It has the property `beta_parameters`, that is, the parameters of the **beta distribution** that model the actual conversion rate. It provides the methods:

– `pull_arm()`: returns the index of the arm that maximizes the beta distribution.

– `update(pulled_arm, reward)`: updates the beta distribution for the `pulled_arm` using the value of the `reward`.

`PricingScenario`: is the class responsible for the simulation of the environment. It provides the methods:

– `get_optimal_arm()`: returns the arm that maximizes the conversion rate.

– `round(pulled_arm)`: play the `pulled_arm` and obtain a reward.

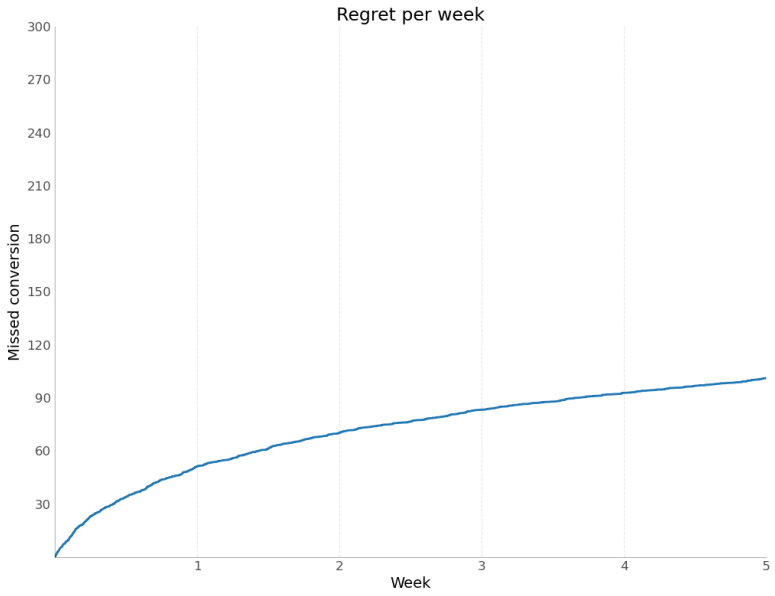

Figure 4.1: Regret cumulative sum per iteration

4.2 Execution

During the execution, what we want to maximize is the total number of converted consumers: to make this computation easier we suppose that the number of consumer reached by playing every arm is roughly the same. In **figure 1.3** (chapter 1), we can see the actual conversion curve. using this data, we can execute our estimation algorithm. Every day we play a different arm $24 * 6$ times, using a Thompson's sampler to learn the learn the best price to play.

Knowing what the ideal gain is, and what the actual gain is, we can easily estimate the regret for every iteration. As we can see in **figure 4.1**, the regret increases very rapidly in the first few iterations, stabilizing later on. This means that the Thompson Sampler has actually learned the conversion rate curve.

Small increases in the regrets are caused by the fact that the sampler doesn't always selects the optimal arm. Meaning that, since there are two different arms with a very similar value, the sampler prefers to explore both the solutions to obtain the best approximation possible and find the actual maximum.

Chapter 5

Pricing with Context Learning

What we want to achieve in this chapter is to identify the optimal context partitioning and the related optimal prices.

5.1 Design philosophy

In this chapter, as in chapter 4 we are simulating a static **environment**: a user sees an ad, clicks on it, and will decide whether to buy the advertised product or not based on its price. We refer to the percentage of users that buy the product at a given price as the **conversion rate** of that price. We want to model the conversion rate for a fixed set of prices, thus modeling a **conversion rate curve** for the advertised object.

Users are split in different classes based on the value of some attribute: in our case we observe their **age** and **sex**. This subdivision of users in groups is referred to as **context**. In this way, we can sold the same item at two different prices to two different groups.

In our model, therefore, we estimate the conversion rate curve for each user class: each context has a (possibly different) curve, and, as a consequence, is associated with a (possibly different) optimal price. With a weighted sum of their conversion functions, we can estimate a conversion rate curve for the whole context.

Our objective is to find the context that maximizes the conversion rate. This can be obtained by executing two different algorithms:

- A **Thompson Sampler** to estimate the conversion rate curve for a given context. This follows the same loop described in 4.1;

- A clustering decision-tree based **context generation algorithm** to find the optimal context. This search is based on the results of the Thompson Sampler in the previous week.

The clustering algorithm uses as criteria to decide whether to split or not:

$$\mu \leq p\mu_{a_1^*} + (1-p)\mu_{a_0^*}$$

Where μ is the information before the split, p is the probability of attribute a of being equal to 1, while $\mu_{a_1^*}$ ($\mu_{a_0^*}$) is the information obtained by playing optimal arm a_1^* (a_0^*) when attribute a is equal to 1 (0).

As discussed in chapter 1, the estimation of the optimal context is based on the presence of a set of two binary attributes. In **figure 1.3** we can see how, while the conversion rate curve isn't affected by the age of the costumers, it's strongly dependent on their sex. It follows that an ideal algorithm would partition based on the consumer's sex. To highlight how much the choice of the right contexts matters, we start from one of the worse situation possible: we split based on the age but not on the sex.

5.1.1 Classes

`Learner`: same as in chapter 4.

`TS_Learner`: same as in chapter 4.

`PricingScenario`: same as in chapter 4.

`ExtendedPricingScenario`: extension of `PricingScenario`, it add the possibility of get selective responses from the environment by selecting the sex and age of the users we are interested in. It provides the methods:

- `get_optimal_arm(sex, age)`: returns the arm that maximizes the conversion rate of the context described by `sex` and `age`.
- `round(pulled_arm, sex, age)`: play the `pulled_arm` and obtain a reward of the context described by `sex` and `age`.

`Context`: representation of a possible partition. It is identified by `sex` and `age` and it contains a `TS_Learner`.

`ContextLearner`: responsible of learning the context based on a dataframe of observations. The recursive function `split()` recursively builds a decision tree.

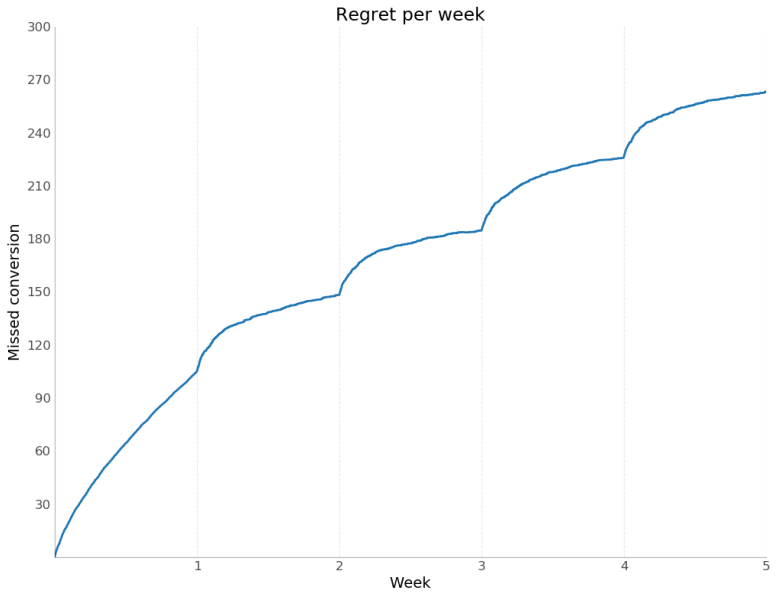

Figure 5.1: Regret cumulative sum per iteration

5.2 Execution

During the execution, what we want to maximize is the total number of converted consumers: to make this computation easier we suppose that the number of consumer reached by playing every arm is roughly the same. In **figure 1.3** (chapter 1), we can see the actual conversion curve. Using this data, we can execute our estimation algorithm.

The core idea is to run the context generation algorithm at the end of every week, using the information gathered in the previous days. Every day we play a different arm $24 * 6$ times, using a Thompson's sampler to learn the best price to play for every context. The data collected during this process is the one used to learn the context.

Knowing what the ideal gain is, and what we actually gain, we can easily compute the regret for every iteration. Crucially, as we can see in **figure 5.1** the regret increases very rapidly in the first week, where the partitions are not ideal.

We can observer that, by the end of week 1 – after having learned a new

partition – the regret doesn't increase as fast. In fact, after few iterations, the actual gain converges to the ideal one. All the following weeks have a similar pace, showing that we have converged to an optimal context partition. As we've predicted at the beginning of the chapter, we obtain an optimal result when we generate two different contexts: one for female and one for male costumers.

Chapter 6

Simultaneous Optimization

The objective of this section is to develop an algorithm capable of finding the optimal budget allocation for 3 different campaigns in order to maximize the total number of items sold when we can choose a different price for every context.

6.1 Design Philosophy

In this chapter we are a simulating a static environment: a user sees an ad, clicks on it, and decides whether to buy the advertised product or not based on its price.

Differently from what has been previously done, here we want to chose the optimal budget allocation for the advertising campaigns in order to maximize the amount of items sold. To do that, we make the assumption that every subcampaing is targeted towards a specific user group –we suppose that this information is known a-priori– and that such user group has an unknown conversion rate curve.

In chapter 2 and 3 we wanted to solve the problem:

$$\begin{cases} \max_{y_j} \sum_{j=1}^{N} n_j(y_j) \\ s.t. \sum_{j=1}^{N} y_j < Y \end{cases} \tag{6.1}$$

Where j is a subcampaign, y_j is the budget allocated to it, n_j the number of clicks and Y is the budget cap.
However, now, we want to solve:

$$\begin{cases} \max_{y_j} \sum_{j=1}^{N} v_j(p_j) n_j(y_j) \\ s.t. \sum_{j=1}^{N} y_j < Y \end{cases} \tag{6.2}$$

Where p_j is the price of the item for the coalition j and v_j is the **value of a click** for the related subcampaign. In our case, the value of every campaign is

the percentage of users who bought the product after clicking on one of the ads of that subcampaign, the so called **conversion rate**.

Notice that v_j is a dependent on campaign j only because we assumed that a subcampaign is targeted only to a specific target demographic. In fact, for every subcampaign and for every budget, we can estimate how many people of a specific user group click on one of its ad, obtaining n_j. Having done that, we can select the pricing for that group in order to maximize the number of items sold, obtaining an estimate for v_j.
Repeating that for every subcampaign, we can solve the **budget allocation problem** (*equation 6.2*) for time t.

This problem is a combination of an advertising problem and of a pricing problem. As such we will need:

- A **Gaussian Process Thompson Sampler** to model and estimate the number of clicks of an advertising campaign targeted to a specific group;
- A **knapsack-like** algorithm to decide how to allocate the budget to the advertising campaign based on the results of the samplers;
- A **Thompson Sampler** to model and estimate the number of items sold for a given price to a specific group.

The loop we will follow at every timestep t is:

1. For every subcampaign, sample the Gaussian process to estimate the number of clicks n_j;
2. For every pricing scenario (*i.e. for every user group*), pull an arm according to the Thompson sampler, obtaining v_j;
3. Run the knapsack algorithm using the tuples (n_j, v_j) and obtain a new budget allocation;
4. For every subcampaign, simulate the behavior of the selected prices in conjunction with the allocated budget;
5. For every subcampaign, update the model;
6. For every pricing scenario, update the model.

6.1.1 Classes

`CostumizablePricingScenario`: extension of `PricingScenario` (see chapter 4) responsible to simulate the pricing environment for a specific user group described by the properties `sex` and `under_30` of its users.

`TS_Learner`: is the class that implements the Thompson Sampler.
It provides the method:

- `pull_arm()`: return the maximum of the beta distribution and the related arm index.
- `update(pulled_arm, reward)`: update the beta distribution with the sampled `reward` for the `pulled_arm`.

`Scenario`: represents an advertising campaign.
It provides the method:

- `round(daily_budget_index)` to obtain a response to the campaign for a given allocated budget `daily_budget_index`.

`SW_GPTS_Learner`: is the Gaussian Process Thompson Learner associated to an advertising campaign and is responsible for the estimation of the conversion rate.
It provides the methods:

- `sample_values()` to obtain the value of the modeled conversion rate for every possible allocated budget.
- `update(arm_index, reward)`, `update_model()` and `update_observations(arm_index, reward)` to update the model based on the result of the sampled arm.

`Optimizer`: is in charge of executing the knapsack-like algorithm.
It provides the method:

- `optimize(sampled_values)`, that select which arm to play for every sub-campaign based on the `sampled_values` and while constrained by the daily budget.

6.2 Execution

Using the data from the user pricing model, we can model a dynamic conversion rate curve for each of the campaigns, as shown in the first chapter. In particular, we can assign to every campaign a budget from 5'000$ to 10'000$ (sampled every 500 dollars). The daily budget cap is set to 27'000$. We recall that on campaign is for women under 30, one is for women over 30 and the last one for males.

For what concerns the pricing, we model 3 different users class, one for each subcampaign. We can assign to every demographic a price ranging from 50$ to 104$ sampled every 6$. The conversion rate of the price for every class is shown in chapter 1.

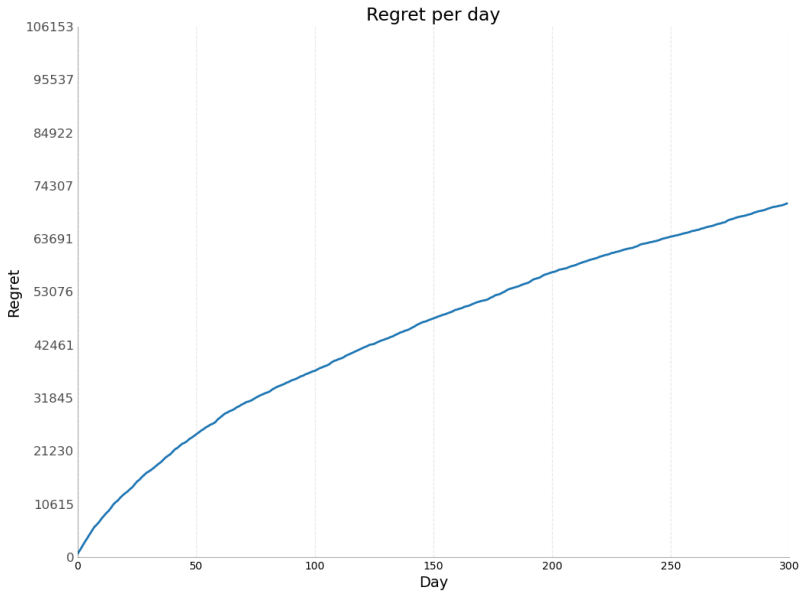

Figure 6.1: Cumulative missed sales

Every day, for 300 days, we run the loop described in **section 6.1**, computing the daily **optimal result**. Differently from previous chapters, the optimal value is related not only to the budget allocation, but also on the price selection. We can see how this impact the execution of our algorithm by looking at the log file `history.log`. The ideal budget is reliably found after 100 days, while the optimal price selection, despite being close to the ideal one, isn't found within our timeframe. This, however, is to be expected since, as shown in the previous chapters, the two problems have different convergence speed: in chapter 2 we saw that it took around 30 40 days to learn the optimal budget allocation, while in chapter four it took more than 3'000 timesteps to converge to the ideal pricing. Unfortunately, due to hardware limitation, it was not possible to simulate the execution of the algorithm over a significantly larger timeframe.

In **figure 6.1** we can see the cumulative regret expressed in terms of missed sales. This is an average over 5 executions of the algorithm, and we did that to obtain a more unbiased estimation of the real curve. As we expected, we can see that the regret grows rapidly in the first few days and continues to do so until the algorithm learns the optimal budget allocation.

As we previously discussed, it is much harder to learn the optimal pricing

and this is reflected by the constant increase of the curve. Notice, however, that the number of missed sales is quite small: in the best case scenario, we can sell up to 940 items per day, while, on average, we will sell around 900 items. This means that we lose only 4% of the possible sales. Moreover, with a longer time horizon this percentage will substantially decrease.

Chapter 7

Simultaneous Optimization with an unique price

The objective of this section is to develop an algorithm capable of finding the optimal budget allocation for 3 different campaigns in order to maximize the total number of items sold when we can choose a unique price for every context.

7.1 Design Philosophy

In this chapter we are a simulating a static environment: a user sees an ad, clicks on it, and decides whether to buy the advertised product or not based on its price. However, while in chapter 6 we could sell the item with a different price for every context, in this case we have to choose a unique price for every user group.
As in chapter 6, the optimal budget allocation is the one that solves the following system of equations:

$$\begin{cases} \max_{y_j} \sum_{j=1}^{N} v_j(p) n_j(y_j) \\ \sum_{j=1}^{N} y_j < Y \end{cases} \tag{7.1}$$

Where j is a subcampaign, y_j is the budget allocated to it, n_j the number of clicks, Y is the budget cap, p is the (fixed) price of the item and v_j is the **value of a click** on subcampaign j (*i.e. the number of people that will buy the item*). Notice that, differently from equation 6.2, here the price p is not dependent on the context but is fixed.

To solve this problem we use the following:

- A **Gaussian Process Thompson Sampler** as in chapter 6;
- A **knapsack-like** as in chapter 6;

- A **Thompson Sampler** as in chapter 6

The loop is very similar to the one of chapter 6. However, in this case, we do not pull a different arm for every context at each iteration. Instead, we are going to play every day a fixed price for all the user groups: step 2 is therefore modified to only sample the beta distribution at the selected price. The loop is the following:

1. For every subcampaign, sample the Gaussian process to estimate the number of clicks n_j;
2. Sample the Beta distributions at the selected price, obtaining for every context a value v_j;
3. Run the knapsack algorithm using the tuples (n_j, v_j) and obtain a new budget allocation;
4. For every subcampaign, simulate the behavior of the selected prices in conjunction with the allocated budget;
5. For every subcampaign, update the model;
6. For every pricing scenario, update the model.

7.1.1 Classes

The classes are the same as in chapter 6, with the exception of `TS_Learner`, where we added the method `pull_arm(arm_index))` that allows us to specify which arm to pull.

7.2 Execution

Using the data from the user pricing model, we can model a dynamic conversion rate curve for each of the campaigns, as shown in the first chapter. In particular, we can assign to every campaign a budget from 5'000$ to 10'000$ (sampled every 500 dollars). The daily budget cap is set to 27'000$. We recall that on campaign is for women under 30, one is for women over 30 and the last one for males. For what concerns the pricing, we model 3 different users class, one for each subcampaign. We can assign to every demographic a price ranging from 50$ to 104$ sampled every 6$. The conversion rate of the price for every class is shown in chapter 1.

Every day, for 300 days, we run the loop described in **section 7.1**, computing the daily **optimal result**. However, the price of the item to sold is fixed and is dependent on the day and can by computed as:

$$daily\_price = price[day\%10] \tag{7.2}$$

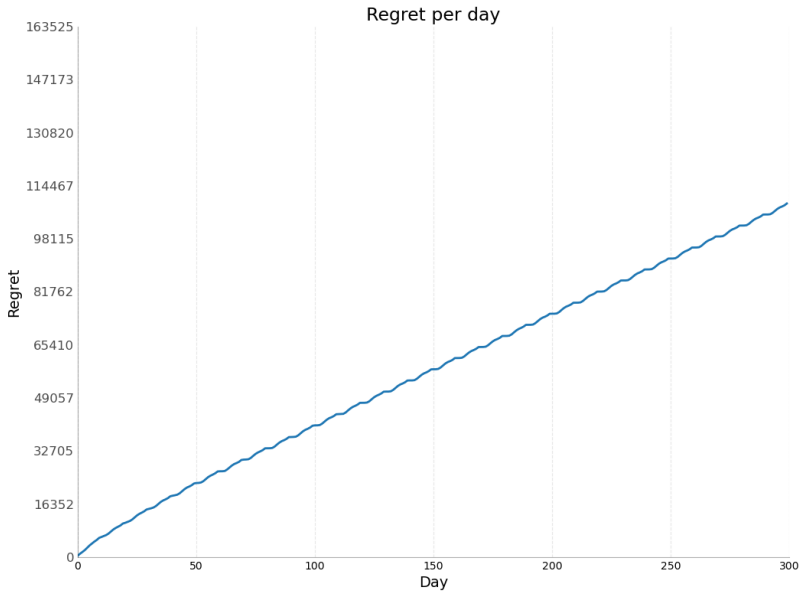

Figure 7.1: Cumulative missed sales

Meaning that, in 300 days, we'll play the same price 30 times. Since the optimal solution is dependent on the price, the regret will always increase: there is no implicit learning of the pricing.

In **figure 7.1** is shown the regret in terms of missed sales. As we can see, the regret has a saw-like behavior: it increases every day the price is not the ideal one and remains more or less constant whenever we play the ideal price. Notice that this happens only after a few iterations, meaning that the algorithm correctly learns the ideal allocation of budgets.
This is confirmed by analyzing the file `history.log`, where we can see the regret, the price and the budget allocation played every day. It shows that, after a few iterations, the correct budget allocation of 10, 10, 4 is learned and that the price of 56$ is the one that maximize the number of total sales.

As a finale note, we want to stress how this algorithm doesn't learn a pricing conversion rate curve, but rather it shows the result of different prices while learning the optimal budget allocation. The selection of the correct price should be done –manually or automatically– based on the results rather than by maximizing the conversion rate function.

Appendix A

Activities

In this appendix we describe how our work has been structured.

The first thing we did, was to organize the workload into 4 different areas: dataset design, pricing, advertising and simultaneous optimization. All were done in parallel, beside the last one, which required the results from pricing and advertising.

Dataset design involved the creation of a dataset that, on one hand, was realistic enough and, on the other hand, had some interesting features. In fact, we built the dataset aiming to highlight how our algorithm behaved. First, we proceeded to generate a simple dataset that enabled us to test the algorithm. Then we reworked it to produce more interesting cases. This work was done at the same time as the work on the pricing and advertising parts.

Advertising includes the work discussed in chapter 2 and 3. The common theme is the design of an algorithm able to decide how to allocate a budget to a set of 3 subcampaigns. This was quite easy to design but more tricky to implement. In particular, the knapsack-like algorithm, while conceptually simple, required a lot of fine tuning and bug fixes to work as intended. This is the first are of the project that has been developed.

Pricing includes the work discussed in chapter 4 and 5. The common theme is the design of an algorithm capable of deciding the best pricing for an item in order to maximize the number of sales. As for the previous topic, it didn't require too much work on the design phase, even if we spent a lot of time on the design of the clustering algorithm for the context generation. Differently from what happened with advertising, we implemented the algorithms without too many difficulties. We developed this area after advertising, and we designed point 5 first, as point 4 was a simplification of it.

Simultaneous optimization includes the work discussed in chapter 6 and 7. The common theme is the design of an algorithm capable of learning the budget allocation and the pricing at the same time. This was, by far, the part that took the most time for the design phase. The implementation went down quite easily, as it required to put together some basic 'building blocks' developed in the previous chapters. On the other hand, the design forced us to make some considerations that were not trivial (e.g *deciding the order of the operations, understand how to structure the budget allocation problem, etc.*). Clearly, this part was done last.