Project Report - Online Learning Applications

Prof. Nicola Gatti

Social Influence and Advertising

POLITECNICO

MILANO 1863

Francesco Puoti [10595640]

Federico Romeo [10566536]

Filippo Rescalli [10669906]

Riccardo Zanelli [10572141]

Contents

1 Introduction

1.1 Context

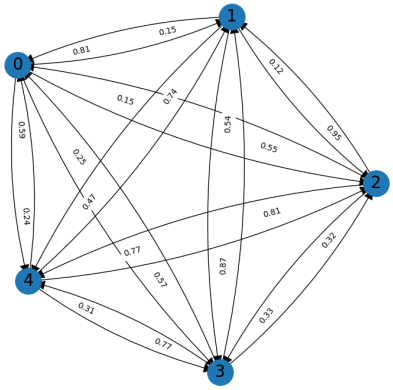
The real-world scenario to be modeled is an advertising problem of an e-commerce company that wants to choose the best budget allocation on 5 different products by taking into account an expense upper bound. The bidding system is assumed to be already in place so no bidding optimization is performed; the only goal is finding the optimal budget to assign to each product's advertising campaign.
Every day a number of customers will land on the product pages managed by the e-commerce. The users interact with the products by browsing the web pages of the e-commerce. If the user is interested in buying the product, he can first specify the number of units and then add it to the virtual cart.
After doing so, two other products are recommended and made accessible through advertising slots. There are three classes of users, each one characterized by a different spending behavior, defined by two binary features. The time is discretized and the reference time step for the implemented algorithms is one day.

Why do we need online algorithms? The goal of the seller is to maximize the profit, so we aim to learn the best allocation of the total budget among the 5 sub-campaigns. We know that varying the amount of budget spent on a campaign will affect the sales of a specific product , but we don't know the exact underlying dynamics and relationships. Given an allocation, we can get real-time feedback from the environment by simply measuring (at the end of each day) the profit generated by the advertising. We can consequently resort to online algorithms to tackle the task of optimizing the allocation.

1.2 Scenario

The variables and parameters defining the problem are generated in the Scenario class, which is used to initialize the Environment object. Specifically these are the core attributes with their respective generation policy; the latter has been chosen to best fit the nature of the attribute itself:

- **network graph**: the underlying structure of the e-commerce website. Every node represents one of the five products and each arc is associated with a weight representing the probability of going from a specific product page to another one. We have to take into account that we have to deal with two different settings of the graph weight: the first one being fully connected, and the second one not, thus having some edges with zero probability.



- **observation_probabilities**: a random matrix representing the probability of observing a secondary product *j* starting from the web page of a primary product *i*. Note that the secondary products are displayed in two slots, displayed one above the other, thus giving more importance to the top one. The observation probability is 1 for the first slot and *LAMBDA* for the second slot. The value of *LAMBDA* is assumed to be fixed at 0.6.

- **product_prices:** an array representing the price of each product. The array containing the value is fixed

- **user_reservation_prices**: define the prices that a user is willing to pay for a specific product. In order to generate the array, we imagined for each an *appreciation* of the users for each product. Therefore the user_reservation_prices array is computed as product_prices + user_appreciations. Obviously, the appreciation can be positive or negative according to the willingness of the users to buy or not. The user buys a number of units of the primary product if the price of a single unit is under its reservation price for that product.

- **user_poisson_parameters**: array containing for each tuple *(user_class, product)* the lambda values defining the poisson distribution of the bought items in the Monte Carlo sampling, later explained.

- **alpha_bars**: *alpha_bars* represents the MAX percentage of users (for each class) landing on a specific product webpage including the competitor one. The array containing the values is fixed.

As mentioned at the beginning, users can belong to 3 different classes. We imagined that for this reason each class has its own *user reservation prices*, since they should have different interests for the various kinds of products sold by the ecommerce, *poisson parameters* and *alpha ratios*, thus simulating different spending behaviors.

Another important attribute in the problem are the **budgets**, which represent the amount of money that can possibly be allocated for a single product campaign. In this context, the allocable amount is assumed to be discrete. Specifically, the budgets array is defined as follows:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Notice that 10 is the maximum that the business can spend in the advertising campaigns, and from now on, it will be referred to as **B_cap**. We opted for this discrete approach since it's easily scalable and it would just take a simple multiplication to male the values continuous and resort to prices from real world scenarios.

1.3 Environment

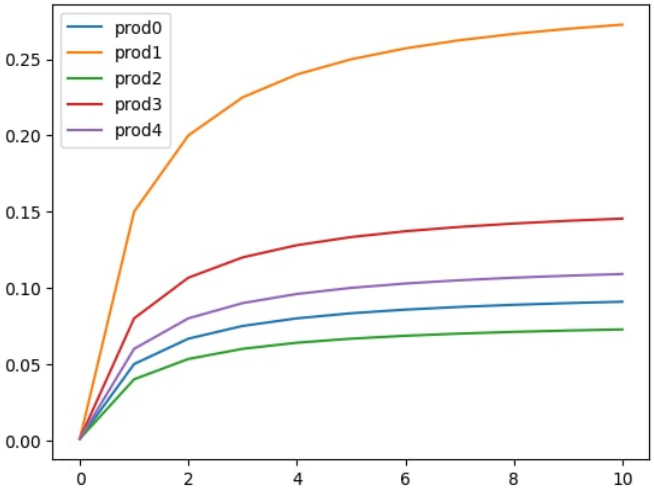
The environment is designed to simulate the interaction between a user and the e-commerce website. It is defined by all the parameters that are outside the scope of the Ecommerce, thus, representing the underlying structure of the problem. The Environment object is initialized by means of the *Scenario* class which is responsible for generating the following parameters: *user reservation prices, graph weights* (probabilities), *alpha bars* and the *poisson parameters.*

Alpha values computation
Apart from containing these information, the Environment is also responsible for the generation of the α ratios, namely the percentage of users landing on a specific product page every day. The value of the α ratios will be realizations of independent Dirichlet random variables as required by the project specifications. A dedicated function has been implemented to return the α ratio for each <*product, allocated_budget*> tuple. The computations proceeds as follows:

1. a mapping function is evaluated for each tuple *(product, budgets)*. The function is defined as :

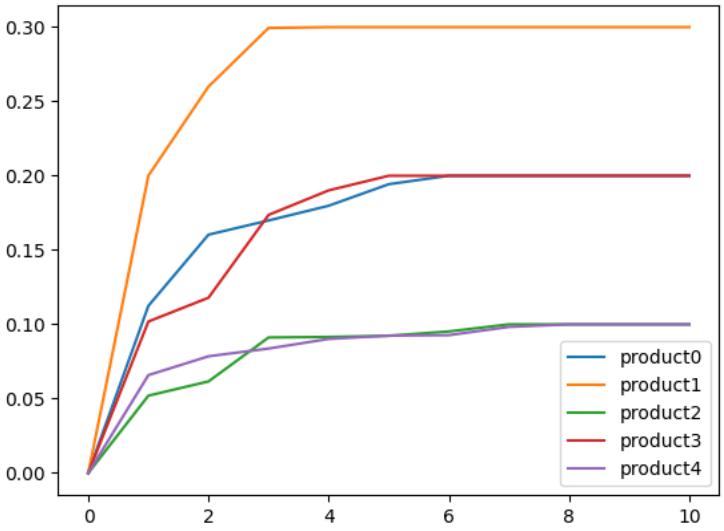
$$f(product,\ budget) = \frac{2\ \alpha[\ product\ ]}{1 + \frac{1}{budget}}$$

and returns a value in the range (0,1).



The higher the budget, the higher the concentration parameter for that product. The shape that we designed for the alpha functions reflects the intuition that after a certain budget value, the output is capped (by the $\bar{\alpha}$). This is a design choice to differentiate the different products, still having similar functions between them. Notice, the competitors' product is supposed to have concentration parameters equal to the mean of those of the 5 products of the e-commerce business.

2. For each user class, all the concentration parameters are fed in the Dirichlet, and the results are saved as the *users alpha* for the day. Below an example of the results of the dirichlet for one day. For the sake of the plot simplicity, these are the 'aggregated alphas', that is the sum over the user classes has been computed, and the alpha of the competitors' product has not been plotted. In fact, we can notice how the sum of the maximum of all the functions is equal to 0.9.



2 Social Influence

As already mentioned, the Environment shapes the context that allows the user to interact with the e-commerce website. In order to simulate the behavior of the users on the webpages some *Social Influence* techniques are adopted.

2.1 Simulation of user interaction

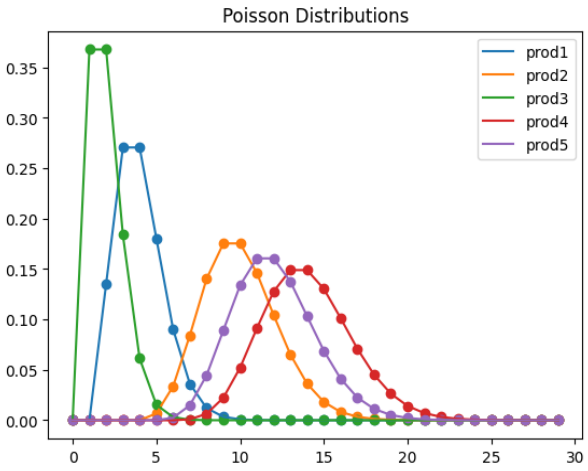
In our code the *Social Influence* class is responsible for performing the interaction simulation. Our class computes each day the *number of items sold* for each *product*. We decided to extract this information in a disaggregated manner. More specifically, starting from each product, the simulation computes the number of items sold of that specific product and of all the others that are reached starting from it. Therefore, the result of this computation will be a matrix of shape (*num_of_products x num_of_products*), where each cell *(p1, p2)* represents the *number of items* of the product p2 sold to those users who landed on the page in which the product p1 is primary.

Computing exactly the *num_sold_items* would require an exponential time in the number of edges, which is not feasible in this problem, even when the product graph is not fully connected. For this reason the class contains two methods that are used to perform the *Monte Carlo estimation* of the activation probabilities.

2.2 Monte Carlo

Monte Carlo is a reinforcement learning technique that allows to learn the model of the environment by interacting with it and then evaluating the information with a frequentist approach. The Monte Carlo method relies on the concept of the **live-edge graph** which can be seen as the resulting activation vector, which keeps track of the nodes activated in a single simulation.
The main idea of the Monte Carlo method is to collect enough live edge graphs experiments to provide a reliable estimate of the activation probabilities, so instead of enumerating all the possible live edge graphs, a “reasonable” amount of them is drawn from a probability distribution.

In our code, the method **`generate_live_edge_graph`** implements the logic. During the simulation the probability of a user buying a product is modeled as a binomial distribution influenced by the user reservation price. The number of items a user will buy is a random variable independent of any other variable; that is, the user decides first whether to buy or not the products and, subsequently, in the case of a purchase, the number of units to buy. *How many objects to buy is modeled by a poisson distribution dependent on a different parameter for each class of users.*



Once the user has bought the items, he observes the secondary product(s), and using the observations probabilities the realization of a binomial random variable simulates if the user continues on the page of the aforementioned products.

Obviously, in order to guarantee a termination of the algorithm, each node can be explored at most once. When no other node is to be explored, the routine returns.

2.3 How many repetitions?

Since Monte Carlo provides an estimation, it's important to define the right number of iterations to be performed in order to obtain a consistent and reliable result. For these settings we can resort to some theoretical guarantees.

Knowing that with probability of at least 1-δ the estimated activation probability of every node is subject to an additive error of $\pm\ \epsilon n$ when the number of repetitions is

$$k = O\left(\frac{1}{\epsilon^2} log(|S|)\ log(\frac{1}{\delta})\right)$$

(Where n is the number of nodes and S is the number of products). In this case we have set $\delta = 0.01$, $\epsilon = 0.03$. In our case, k = 8235.

3 Optimization problem

The goal of this step is solve an optimization problem where the objective function is the profit, defined as the difference between the expected margin and the money spent in advertising. In this setting all the environment variables are assumed to be observable.

3.1 Ecommerce class

For this step we introduce the **Ecommerce** class, which is used to model the parameters define for the e-commerce company, for example:

- *B_cap*: the maximum budget available to be spent on advertising campaign, that needs to be optimized in this step; arbitrarily set to 10
- *budgets***:** the possible range of budget values assignable to each campaign.
- *product_prices***:** the prices of the product sold by the e-commerce

3.2 Reward and value per click

In order to run the optimization algorithm, we first need to compute the expected reward for each <*product, allocated_budget*> tuple, this makes it possible to compare different budget allocations. These are the steps performed to compute the reward:

1. Compute the *"expected alpha"* for each specific <*product, allocated_budget*> tuple (because the alpha value of each product changes accordingly to the budget invested in the campaign for that product)

2. Along with a quantitative information on the alphas, we also need to establish the value associated to each click, in order to move from the visits to the money domain, the ***value_per_click*** associated to each product is computed as follows:

   $$value\ per\ click\ =\ (n\ items\ sold)\ \times\ (product\ price)$$

Where the number of items sold is coming from the Monte Carlo simulation.

3. At this point we have all the information to compute the **expected reward** which is computed in the following manner:

$$expected\ reward\ = (value\ per\ click)\ \times (expected\ alpha)$$

3.3 Optimization algorithm

Given the budgets, the determined bidding strategy and the expected rewards, the goal is to find the optimal budget allocation among all the 5 campaigns. The algorithm suited for this situation is known in literature to be an efficient solution to the "Multiple Choice Knapsack Problem". The implementation in this project settings works as follows.

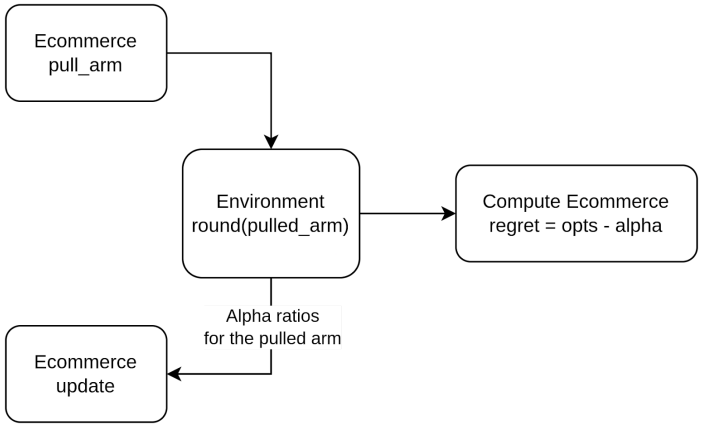
Given as an input table with the different advertising campaigns allocations along the rows and the values of the daily budget along the columns, in which each cell represents the *expected reward* for that specific combination, the algorithm builds a new table by progressively taking into consideration one sub-campaign at a time. The value in each cell of the new tables represents the maximum expected revenue when the budget y_j is distributed among the considered sub-campaign C_i and the previous row. Once the table is expanded considering all the sub-campaigns, the optimal solution is found by taking the maximum in the last row. We can trace back through the table to find the path of choices that led to the optimal solution. The result corresponds to the optimal budget allocations for the sub-campaigns.

The logic just described is implemented in the following methods: `dynamic_knapsack_solver`, `choose_best` and `compute_table` all contained in the *Ecommerce* class.

4 Optimization with uncertain alpha ratios

Differently from the section above, the ratio of users visiting the page with a certain product Pi as primary, called α_i, in this step is unknown and must be estimated. α_0 represents the percentage of users landing on the webpage of the competitor.

This is common in real-world scenarios where the amount of people landing on a website page can't be known a priori, but it is to be estimated in order to better optimize the business. To properly estimate these probabilities two Multi Armed Bandit algorithms are used: **UCB-1** and **Thompson Sampling**.



At the start of the day, the learner has to choose between arms looking at the rewards of the arms played in the past to make the choice of the arm to play. Over time, the learner's aim is to collect enough information about how the rewards relate to each other, so that it can predict the next best arm to play. Each learner is composed of a Gaussian Process to estimate alpha functions and an arm for each budget value. At each iteration a feedback corresponding to the selected arm is given to the environment. Now we update the regressors estimation of the alpha functions and compute a realization of the functions for the selected budget values.
Last step regards conversion of alpha functions in a revenue estimation and to achieve it we assume that the behavior of the user during the interaction can be studied according to the social influence dynamics.

4.1 Detailed information about the Learners

For start, each of the learners is composed of one Gaussian Process Regressor per product. The GaussianProcessRegressor from the python library sklearn

has been used together with the Matern Kernel. The latter has been chosen because it is a generalization of the RBF kernel. In fact, using the parameter *nu* we were able to control the smoothness of the learned function. By setting it to infinity is equivalent to using the RBF kernel.
The gaussian process regressors are in charge of learning the alpha functions, capturing the uncertainty inherent in the Dirichlet probability distribution.
In order to pull the arm, at every round the optimization algorithm is run on a table computed by multiplying the alpha estimation of the gaussian processes and the number of items sold known from the environment.

- **UCB**: maximizes arm means + upper confidence bound.
  Differently from the classical UCB-1 algorithm, the Hoeffding bound has been substituted by a **Bayesian bound**: $\sqrt{\frac{2\,log(t)}{n_a(t)}} * \sigma_a$
  The Hoeffding bound is multiplied by the variance estimated by the gaussian process, in order to put more attention on those arms for which the collected rewards are not very stable. The Bayesian bound works very well in practice, especially in this setting where the distribution to model is a Dirichlet, also known as Multivariate Beta Distribution.

- **Thompson Sampling**: maximizes the samples drawn out from the gaussian process estimated distribution:

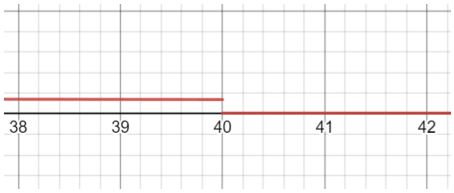
```
samples = gaussian_process_regressor.sample_y(arms)
```

  Noteworthy is that in order to avoid overfitting and the Thompson Sampling to be stuck in a local optimum, we set an exploration probability. At each round:

```
if binomial(1 - exploration_probability):
      pull arm exploiting knowledge
else:
      pick rand super_arm complying with the combinatorial constr
```

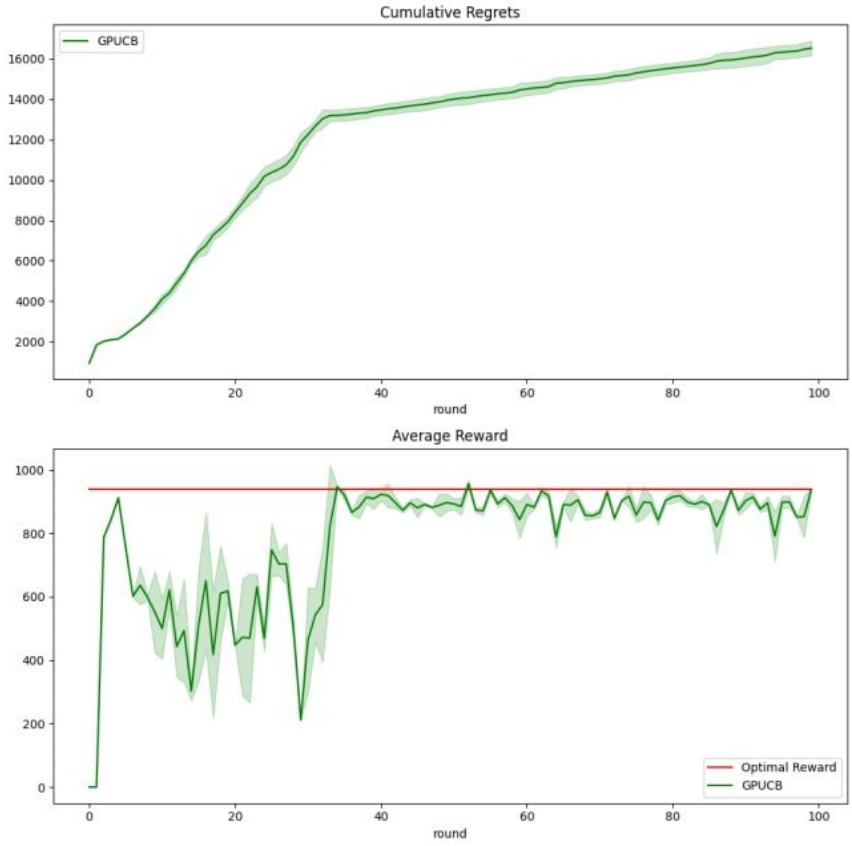
  The exploration probability is dependent on the number of samples observed, since we can assume the higher the sample observed the more precise will be the acquired knowledge:

```
if t < 40 :
      exploration_probability = 1/sqrt(t +10)
else :
      exploration_probability = 0.01
```

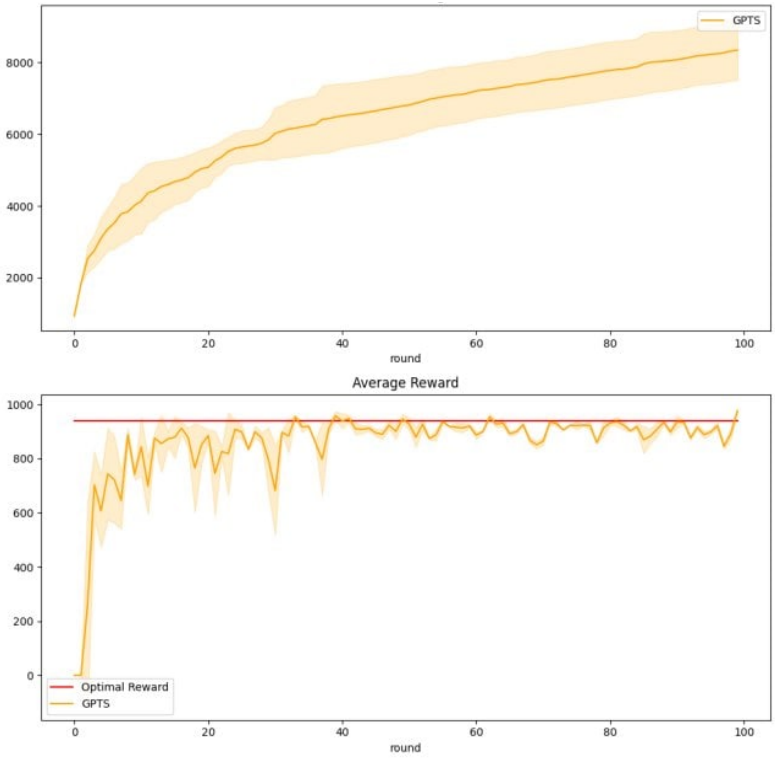


4.2 Results

UCB



Thompson Sampling



4.3 Comparison

Both the algorithms have achieved a logarithmic regret. This is in line with theoretical upper bounds for the two algorithms, which are:

- $R_t(UCB) \leq \sum_{a:\, \mu_a < \mu_{a^*}} \frac{4\, log(T)}{\Delta_a} + 8\Delta_a$

  where Δ_a is the difference between the expected rewards of the optimal arm and the suboptimal ones

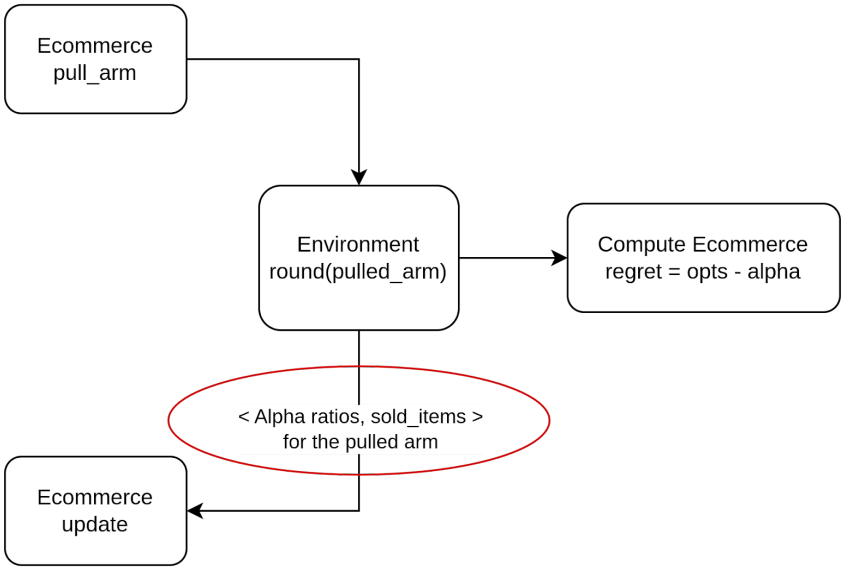
- $R_t(TS) \leq \sum_{a:\, \mu_a < \mu_{a^*}} \frac{\Delta_a\,(\, log(T) + log(log(T))\,)}{KL(\mu_{a^*}, \mu_a)} + C(\epsilon, \mu_{a_1}, \ldots, \mu_{a_{|A|}})$

  where ϵ is any strictly positive value close to zero, C is a constant depending on ϵ and the expected reward of the arms and KL is the Kullback Leibler divergence.

In addition, we can notice how the UCB approach reached the optimal solution less smoothly than the Thompson Sampling. This is because the UCB explores a lot thanks to its upper bounds, which depends most on the variance of the arm. And, in fact, we can observe a sharp cut between the exploration phase of the UCB and its exploitation.

5 Optimization with uncertain α-ratios and number of items sold per product

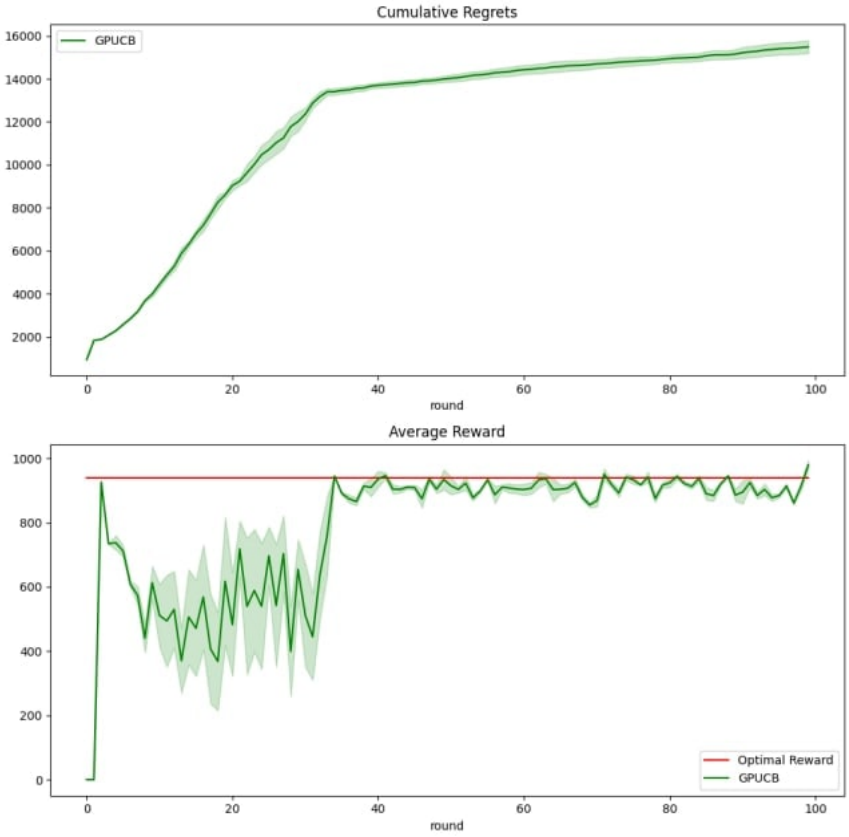
In this scenario, in addition to the *alpha-ratios*, even the *number of items sold* for each product are unknown. The number of items sold is crucial because a product of which many items are sold will have a higher reward with respect to a product sold in a low quantity. This seems realistic since the learner doesn't know how many items will be sold tomorrow. The best estimator in this case is the mean of the observation.
In fact, in this step we deployed the same learners as in the previous case, plus the additional **SoldItemsEstimator class**.
The latter keeps track of the average observed number of items sold.



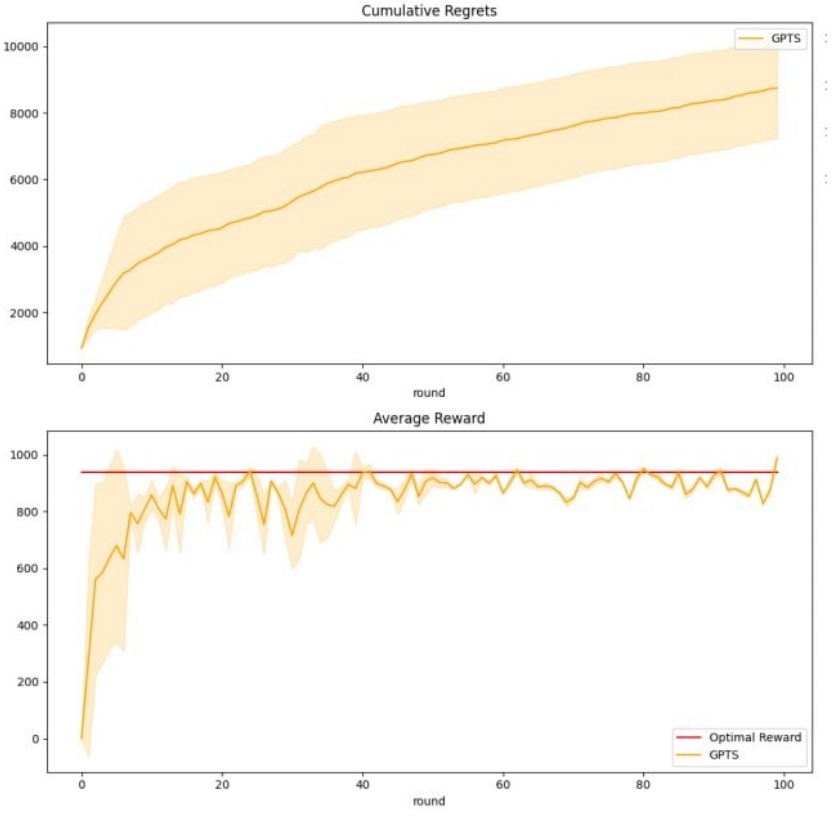
The *sold_items* for the current alpha ratios are computed by multiplying the sold items (returned by the social influence simulation) by the ratio between the *alpha* of the pulled arm and the *alpha bars*.

5.1 Results

UCB



Thompson Sampling

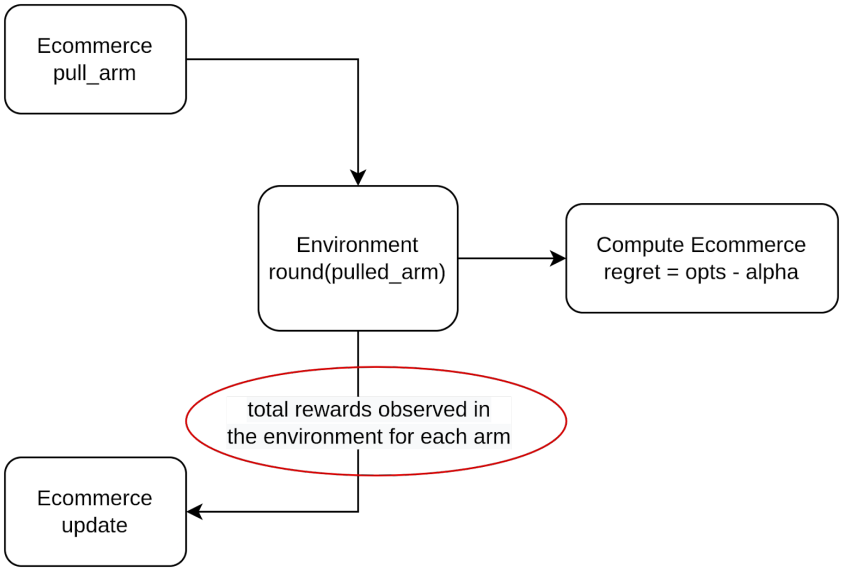


5.2 Comparison

The considerations on the algorithms are the same as in the previous step. What we can notice is that, specially for the Thompson sampling, the standard deviations of both the regret and the collected rewards are much higher than that in the step above.

6 Optimization with uncertain graph weights

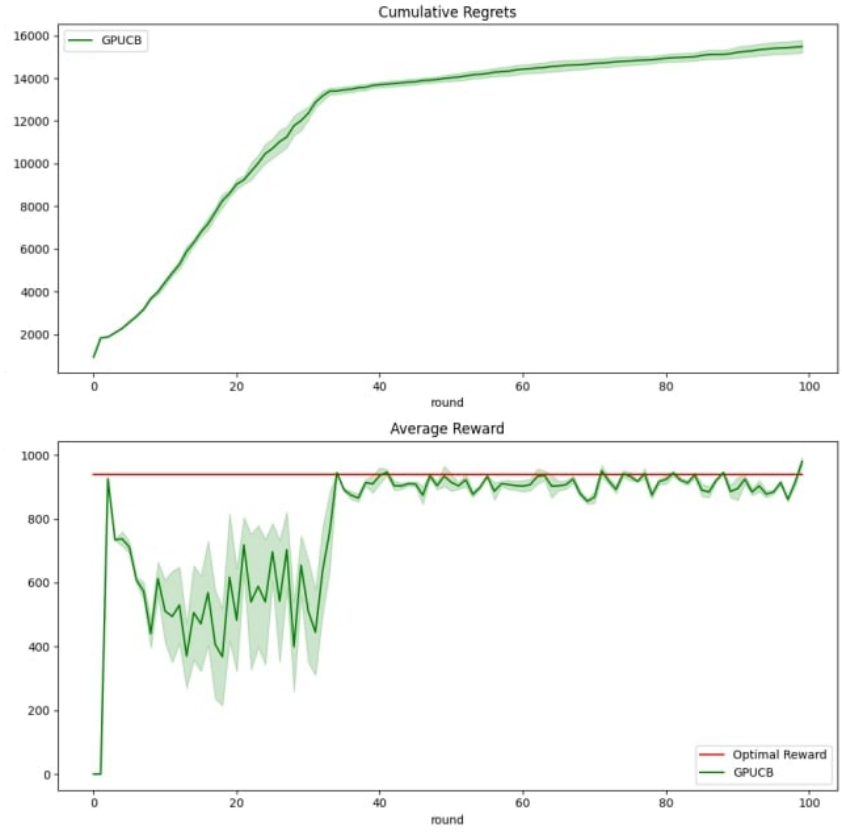
In the new scenario only the graph weights are unknown, meaning that the learner doesn't know the probability of a user reaching the secondary products. Since the *Ecommerce* does not have the information about the value per click, the estimated reward will directly be the samples drawn by the gaussian process.



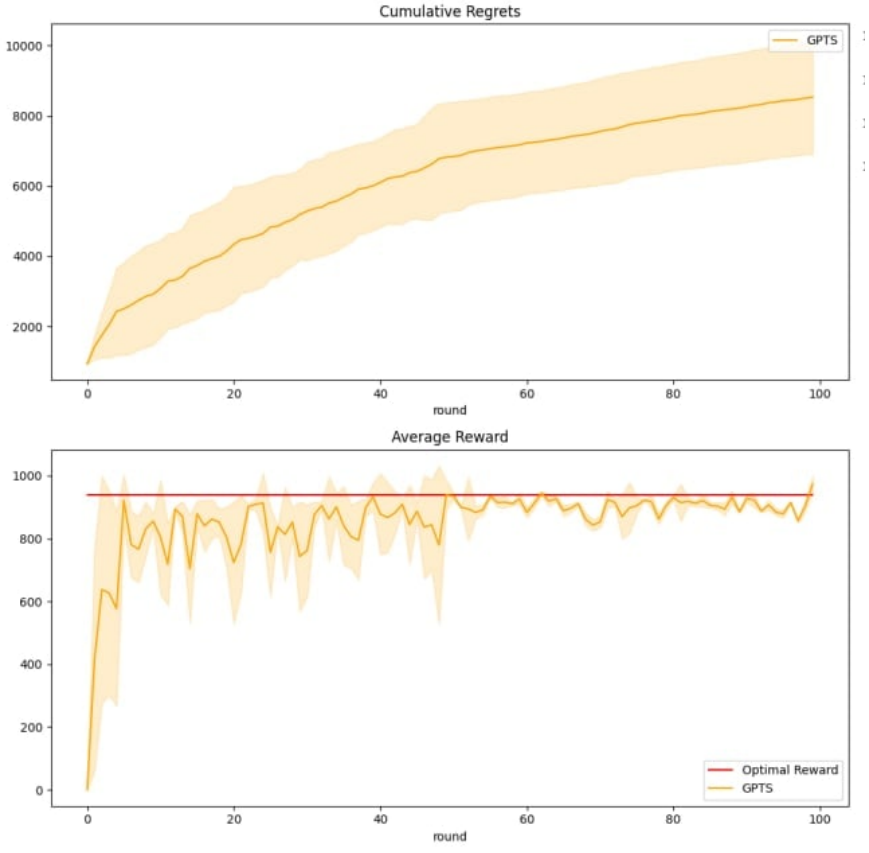
Not having the graph weights means that we cannot leverage on social influence techniques to compute the number of items sold. The Ecommerce in this case is forced to directly model the distribution of the total reward observed for each arm.
The learners are exactly the same as in the scenario described in section 4.
Differently from the scenario of section 5, we do not have any sold items estimator. In fact, the estimation is aggregated into the reward given by the environment.
Note on the Thompson Sampling in this step the exploration has been a little bit increased with respect to the base learner. Firstly, the number of samples needed to reduce the exploration to 0.01 has been increased from 40 to 50. Secondly, the exploration probability in the very earlier phases is $\frac{1}{\sqrt{t+2}}$, higher than the ones in the base learner.

6.1 Results

UCB



Thompson Sampling



6.2 Considerations

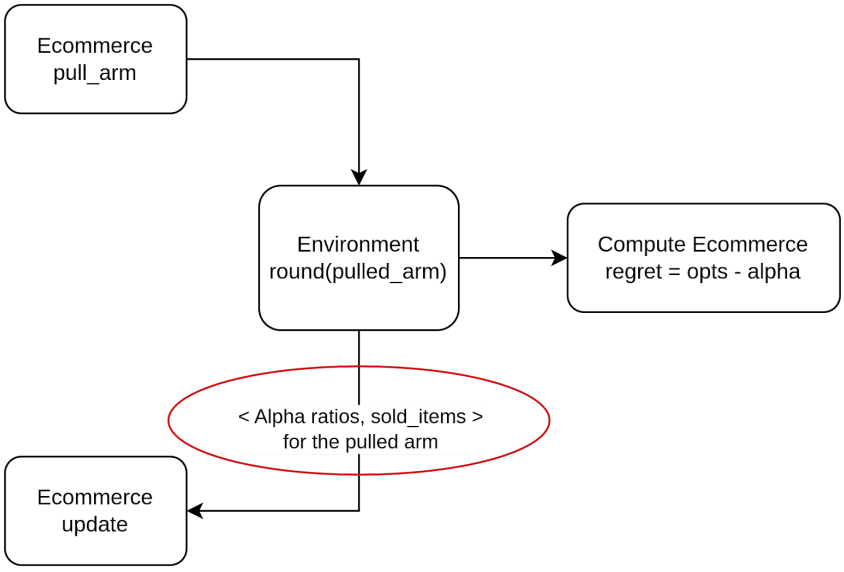
The choice that we made regarding what the e-commerce needed to estimate was not the only one admissible. Another possibility was to let the algorithms learn the distribution of the number of items sold, and then to pull the arm leveraging the information about the alpha rations provided by the environment. But this solution does not seem to be reasonable in a real case scenario. Therefore, we decided to make the environment to aggregate all the information regarding users alphas and number of items sold. In this way the ecommerce, observing the result of its choice, modeled directly the distribution of the Reward function for each arm. Furthermore, the comparisons between the algorithms are the same as in the steps above.

7 Optimization with non stationary demand curve

In this scenario, the probability distribution of the random variable associated with the reward of each arm may change over time. The changes considered in this step are *abrupt changes*, which are the most difficult to handle since they register big sudden changes. We can cope with this problems with two different approaches:

- A passive approach: *sliding window*
- An active approach: *change detection*

The simulation has been divided in three phases, with one specific environment each. The *NonStationaryEnvironment* class keeps the information about the current phase. It is composed of a list of three different environments, and at each round it pulls the right one, according to the information it has.



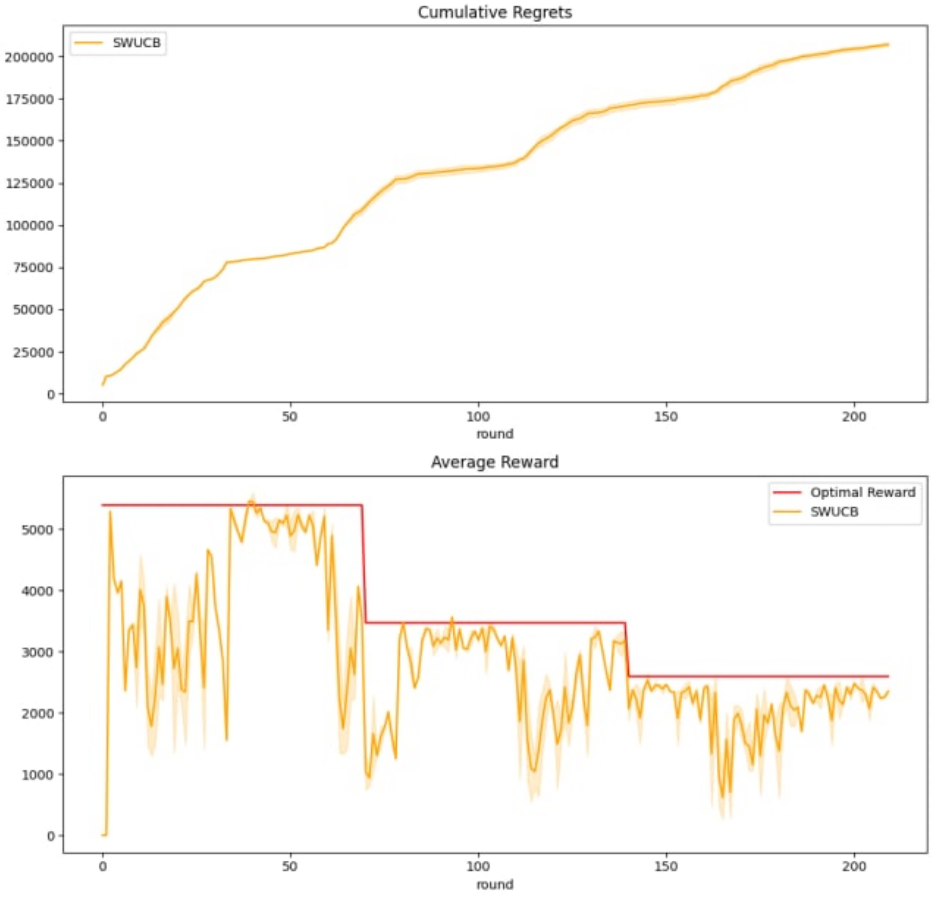
The UCB classes were the same as in the scenario with uncertain alpha functions and number of items sold, apart from the sliding window and the change detection algorithm explained below.

7.1 UCB - Sliding Window

In order to obtain more realistic confidence bounds, the latter are calculated using only the latest τ samples, forgetting in fact the samples too far back in time, that in stationary environments are more likely to hold obsolete information. The size of the sliding window *tau* depends only one the time horizon T as follow:

$$\tau = 2\sqrt{T}$$

7.1.1 Results



The regret in this case is much higher than those in the step4, but it is logarithmic if we take each phase singularly. And this is reasonable, given that for abrupt changes scenarios the regret is:

- given *m = number of breakpoints* = $O(T^{a})$, with $a \in [0,\ 1)$
- apart from the logarithmic terms, $R_t = O(|A|\ T^{\frac{1+a}{2}})$, where |A| is the total number of arms.

We set the number of breakpoints m = 3, since it has to be small with respect to T. Notice how as $\alpha \to 1$, the regret tends to be linear in T.

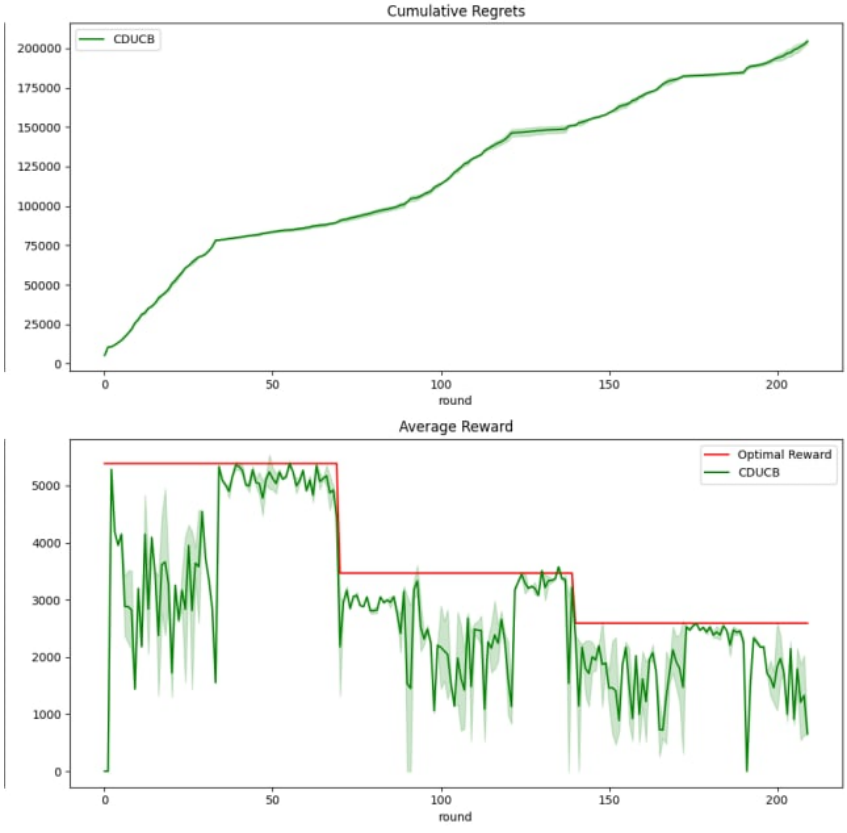
7.2 UCB - Change Detection

The change detection algorithm uses statistics to monitor the drift from previous mean and detect abrupt changes. Each arm is associated with the CUSUM class.

7.2.1 CUSUM algorithm

When a product with a specific price is bought, the CUSUM class is updated for that arm: for the first M times, the function computes the empirical mean over the first M valid samples, this will be considered as the new reference point.
For the successive M+1 samples, it checks whether there has been a change considering the positive and negative deviation from the reference point computed above and the consequential cumulative positive and negative deviation. If such quantities (minus a confidence quantity ε) exceed a threshold *h*, then the change detector will notify an abrupt change to the learner. Once the UCB algorithm has been warned, the valid rewards accumulated so far are discarded, the CUSUM class associated with the arm resets and the learning process restarts. From now the mean and the widths associated with the concerned arm depends only on the valid reward from last change detection onwards. [Reference to the CUSUM algorithm]

7.2.2 Results

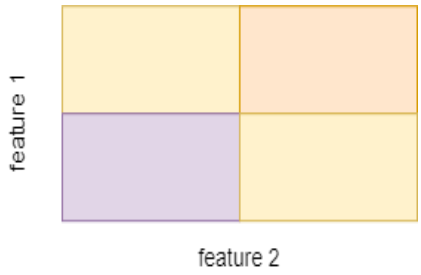


7.3 Comparison

Change detection algorithm (CD) works better than sliding window (SW). This seems reasonable to us since the SW needs τ iterations to eliminate the old rewards while the CD just detects the change and it eliminates the old rewards, starting learning right away from zero.
However, the sliding window approach seems to work better in the long run. In fact, as one could notice from the reward plot, the Change Detection algorithm detects some changes into the environment which are not actually true. This behavior is not unreasonable, since the change detection algorithm is a probabilistic approach. It works by estimating mean and variance of the samples. One could try to fit better the model by tuning the sensitivity parameter h and the confidence ε. Instead, once the sliding window algorithm has forgotten the samples collected in the previous phase, it starts to learn the new distribution, and we can notice how the trend in the collected reward is kept.

8 Context Generation

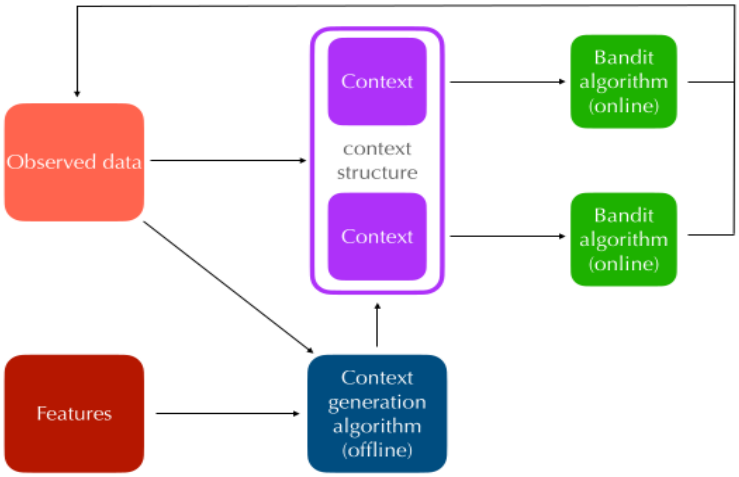
Up to this moment, we considered all the customers interacting with e-commerce as part of a unique group, and thus no features have been taken into account in order to discriminate between different spending behaviors.
The algorithms of the previous steps did not make any distinctions between user classes, and all the algorithms have been performed using aggregated data, but when dealing with multiple classes of users the goal is to exploit this information to fit a dedicated model, i.e. possible disaggregated advertising strategies, for each possible type of user.
Given the problem specifications, each user group is characterized by two binary features, this allows to obtain at most $2^2 = 4$ different classes as combinations of the features. In this problem we consider only 3 different user classes that can be combined as shown in the picture.



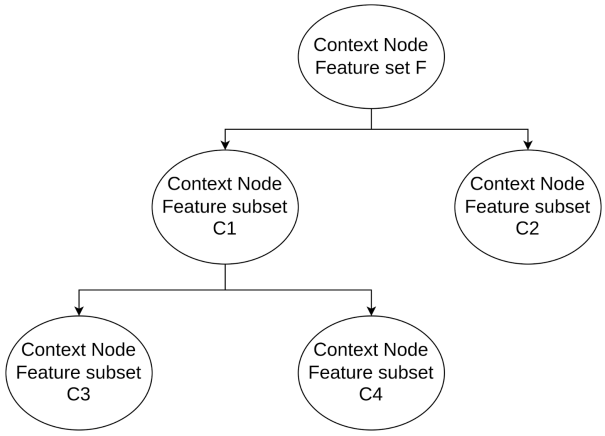
8.1 What is a Context?

Contexts are described as a subspace of the space of attributes with respect to some specific features' values. Using the concept of context to divide the space of the features allows the program to be more efficient in terms of predicting the preferences of the users according to each specific class. During time, when a new context is created, a new on-line learning instance is trained on the old available data that is related to the context itself, and will be used to describe the users having these characteristics from now on.

8.2 Context Generation Algorithm



The main idea of our algorithm is that every 14 days, a context generation algorithm is run to evaluate a splitting condition.
Let's start from the data structure to better understand how the algorithm works.
The *Ecommerce* has a Context Tree, where each Context Node has as attributes the algorithm used and the features the algorithm is analyzing. Obviously, at the beginning, the Context Tree will be composed of just the root node, with the algorithm receiving all the data aggregated.



When the time comes, the splitting conditions are evaluated: starting from the leaves of the tree, for each node:

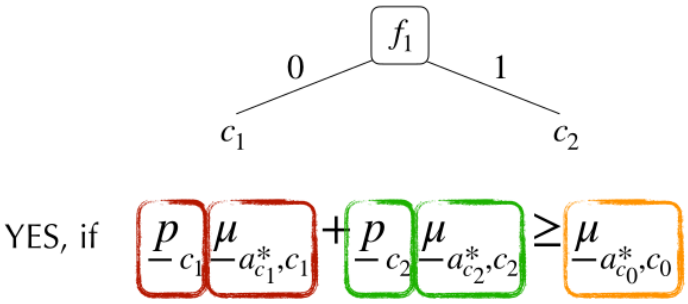
```
for c1, c2 in splits(node.features):

    context_arms, context_reward = compute_context_data(c1, pulled_arms, collected_rewards)
    alg1 = ContextNode().train_offline(c1, context_arms, context_rewards)

    context_arms, context_reward = compute_context_data(c2, pulled_arms, collected_rewards)
    alg2 = ContextNode().train_offline(c2,, context_arms, context_rewards)
```

The function compute_context_data keeps as input a context, the history of the arms pulled and the rewards collected by the ecommerce from the beginning, and it returns the arms and rewards of the specified context.

For instance, assuming that the context C1 describes the first user class, the compute_context_data will return the collected rewards due to the first class of users, and the arm pulled for the advertising campaign targeted to the class.
The split is kept if and only if:



where, starting from left to right, the terms represent:

- Lower bound on the probability that context c1 occurs
- Lower bound on the best expected reward for context c1
- Lower bound on the probability that context c2 occurs
- Lower bound on the best expected reward for context c2
- Lower bound on the best expected reward for the non splitted context

The lower bound on the expected rewards are computed using the Hoeffding bound. More specifically, after having the 2 new nodes trained offline, we keep:

$$\overline{x} - \sqrt{\frac{-log(\delta)}{2\,|Z|}}$$

where:

$\overline{x}$ is the expected reward of the best arm;
δ the desired confidence;
|Z| is the number of data analyzed

8.3 How is the optimization problem solved?

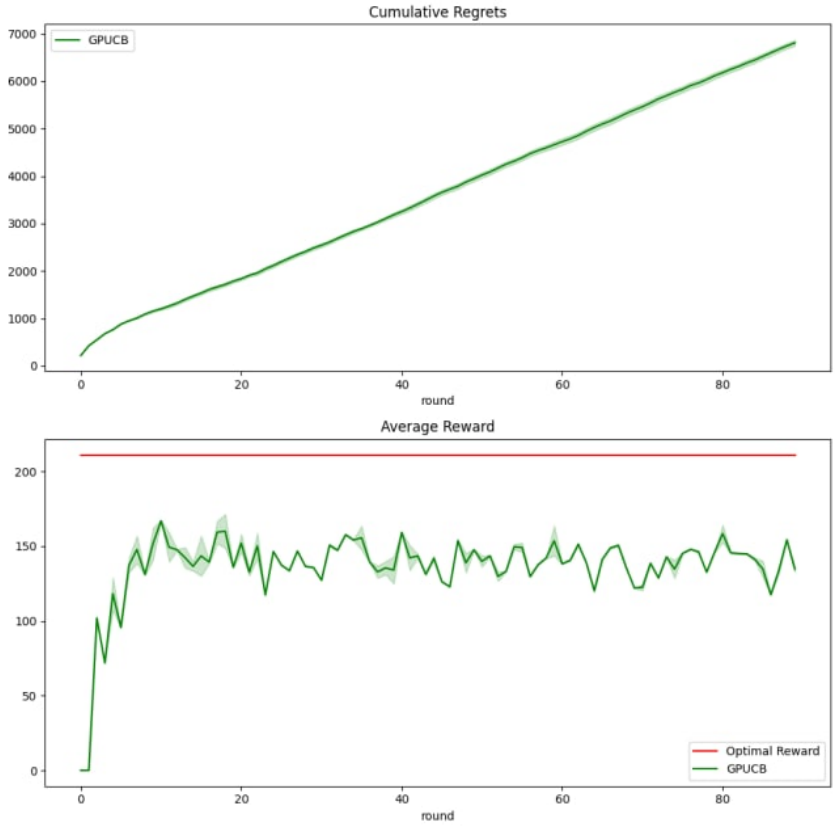
```
for learner in ContexTree.leaves():
      context = learner.get_context_features()
      estimated_alpha[context] = learner.get_estimation()
      estimated_sold_items[context] = learner.get_sold_items_estimation()

pulled_arm = solve_optimization_problem(estimated_alpha, estimated_sold_items)
```

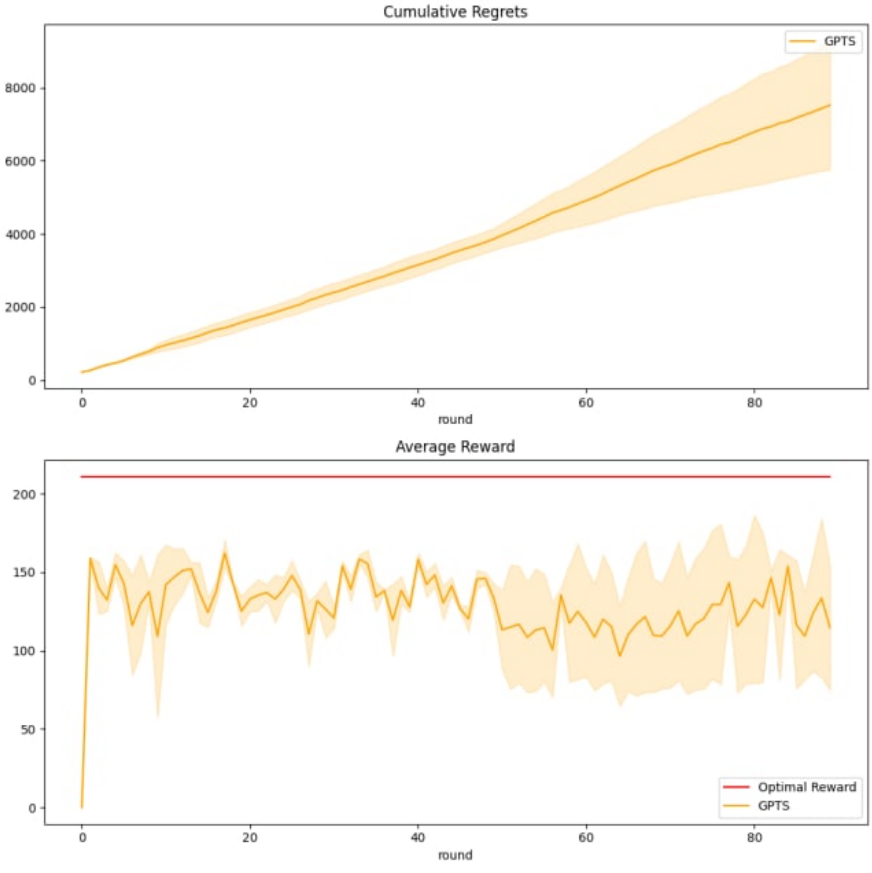
Notice: the optimization problem is run on a table composed of an advertising campaign for each tuple (user_class, product). In our case, 3 user_classes x 5 products = 15 advertising campaigns. Then, when the reward is received by the environment, each of the context nodes will receive the reward just of its context.

8.4 Results

UCB



Thompson Sampling



8.5 Comments

The results of this step were not so satisfying, since both algorithms did not end up with the optimal arm. Despite an initial phase of learning, in which they tend to higher reward solutions, they both get stuck in a local optimum, thus not reaching the global one. By tuning some hyperparameters, such as the confidence δ in the bounds equations, better results could be achieved. What we noticed during the simulation, is that the Thompson Sampling algorithm manages to better split the context than the UCB like approach. This is a reasonable behavior since the UCB grounds its exploration-exploitation trade-off in the variance σ of each arm. Therefore, often the UCB bounds of the non-splitted context happened to be greater than those of the child nodes, preventing the splitting condition to be evaluated as true.

9 Hyperparameter Tuning

Along with the optimization performed on the various steps on the project, we noticed that hyperparameter tuning was really needed. At first we started by adjusting the most important parameters by hand, by trial and error, but we concluded that we needed a more deterministic way to refine them, since they were numerous and their impact on the outcome was huge.

9.1 Genetic Algorithms

We decided to implement some genetic algorithms, both from scratch and from an official library.
First we implemented the classical routine of genetic algorithms, composed of initialization, fitness assignment, selection, reproduction (crossover, mutation), replacement and termination. Since the routine was not easily scalable, we ended up using the *scipy* library function differential_evolution, which was a good fit for our purposes.

The optimized parameters were:

- *length_scale* and *length_scale_bounds* for the Matern Kernel
- *alpha* of the Gaussian processes

We explored the following bounds:

```
alpha_bounds = (1e-7, 1e-3)
rbf_length_scale = (1e1, 1e2)
rbf_length_scale_lower_bound = (1e-3, 1e3)
rbf_length_scale_upper_bound = (1e-3, 1e3)
```

with initial population size set to 15, mutation probability set to 0.5, and crossover probability to 0.7.

9.2 Other tunings

Other crucial parameter that we explored were:

- exploration probability of the algorithms
- In the environment, the concentration parameters of the dirichlet were multiplied by a constant in order to reduce the noise across different sampling. We called this constant *variance_keeper*
- The multiplicative factors of the standard deviation σ in the bounds computation of the UCB algorithms. We came up with the conclusion that the best factor for our scenarios were $\sqrt{2\ log(t)}$