Online Learning Application's Project

Social Influence and Advertising

Raffaello Fornasiere & Christian Spano

September 21, 2022

Useful links and info

- Github Repository: https://github.com/SpanoChristian/OLA-Project
- Project's guidelines can be found here
- We tried to complete step 7 however we got stack into a point and the time was not enough anymore. Therefore, unfortunately, we did not complete it.

The scenario

Imagine an e-commerce website which can sell an unlimited number of units of 5 different items without any storage cost.

In particular, our e-commerce sells the following products:

- Computers
- Smartphones
- Headphones
- Tablets
- Chargers

There are 5 advertising campaigns, one per product, and, by a click to a specific ad, the user lands on the corresponding primary product.

Products and relation graph

- In every web-page, a single product, called **primary**, is displayed together with its price.

Products and relation graph

- In every web-page, a single product, called **primary**, is displayed together with its price.
- The user can add a number of units of this product to the cart.

Products and relation graph

- In every web-page, a single product, called **primary**, is displayed together with its price.
- The user can add a number of units of this product to the cart.
- After the product has been added to the cart, two products, called **secondary**, are recommended in two slots, one above the other, thus providing more importance to the product displayed in the slot above.

Products and relation graph

- In every web-page, a single product, called **primary**, is displayed together with its price.
- The user can add a number of units of this product to the cart.
- After the product has been added to the cart, two products, called **secondary**, are recommended in two slots, one above the other, thus providing more importance to the product displayed in the slot above.
- If the user clicks on a secondary product, a new tab on the browser is opened and, in the loaded web-page, the clicked product is displayed as primary together with its price.

Products and relation graph

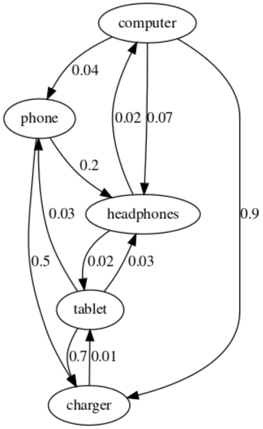
- In every web-page, a single product, called **primary**, is displayed together with its price.
- The user can add a number of units of this product to the cart.
- After the product has been added to the cart, two products, called **secondary**, are recommended in two slots, one above the other, thus providing more importance to the product displayed in the slot above.
- If the user clicks on a secondary product, a new tab on the browser is opened and, in the loaded web-page, the clicked product is displayed as primary together with its price.
- At the end of the visit over the e-commerce website, the user buys the products added to the cart.

Products and relation graph

We represented the likelihood of a user clicking on a secondary product with the following directed-graph

Examples

If a user buys a PC then it is very likely he/she will buy a charger too. For tablets and phones, instead, the probability becomes lower because it is more likely that they already have one at home.



Graph Transitions Learning

- The behavior of the user in the graph is similar to that of the social influence. Thus, we resorted to social influence techniques to evaluate the probabilities with which the user reaches the web-page with some specific primary product.
- What we did is to run a simulation for a certain number of episodes in order to simulate the behaviour of the user (e.g., if he lands on the computer web-page then see which nodes will be activated according to the probabilities)
- From the simulation we then extracted the corresponding probabilities.
- Before running the next algorithms, we run this procedure for some time in order to learn all the transitions weights.

New customers behaviour and α-functions

- Every single customer can land on the web-page in which one of the 5 products is primary or on the web-page of a product sold by a (non-strategic) competitor.
- We call α_i the ratio of customers landing on the web-page in which product P_i is primary, and call α_0 the ratio of customers landing on the web-page of a competitor
- The α ratios are realizations of independent Dirichlet random variables.

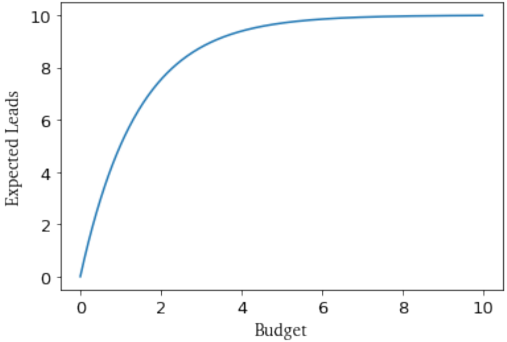
New customers behaviour and α-functions

- Our e-commerce website has a budget B to spend to advertise its products.
- The change of the budget spent on the campaign for the product P_i changes the expected value of the corresponding α_i, and therefore the number of users landing on the web-page in which product P_i is primary.
- For every campaign, we have a maximum expected value of α_i ($\overline{\alpha_i}$) corresponding to the case in which the budget allocated on that campaign is infinite.
- Therefore, for every campaign, the expected value of α_i will range from 0 to $\overline{\alpha_i}$, depending on the actual budget allocated to that campaign.

New customers behaviour and α-functions

To represent the behaviour just explained, we decided to adopt an exponential function:

$$f(x) = \overline{\alpha} \cdot (1 - e^{-x \cdot s}) \tag{1}$$



The environment (1)

Our e-commerce is based on an **simulated environment** which consists of 3 different classes of users distinguished by two different features. We assumed, for our scenario, to use the following two features:

- 1st feature: $f_1 = \{Male, Female\}$
- 2nd feature: $f_2 = \{Young, Adult\}$

The environment (2)

The choice we made, defines three (four) different class of users:

- Young and Male
- Young and Female
- Adult and Male
- Adult and Female (*)

We do not consider the last class of users, (*), since it is quite unlikely, statistically speaking, that this kind of users will reach our web-site.

The environment (3)

- Each of the aforementioned classes is characterized by a profile of α functions, one per campaign.
- This means that, given a campaign, the three classes may have different α functions.

In our scenario, we have

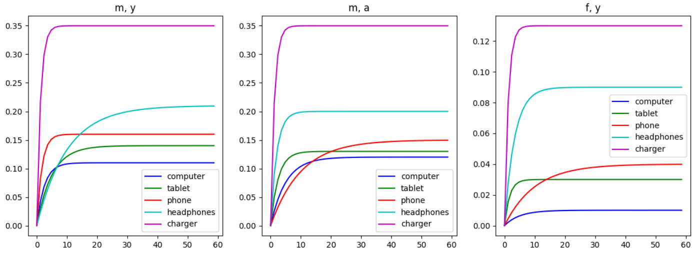


Figure: α-functions of each subcampaign for each class of user.

Our solution (1)

What we did is to build a father Environment class which is a generalization of all the "sub-environment" used in the different steps. Here we report the implementation from an high level point of view (using an UML Class Diagram)

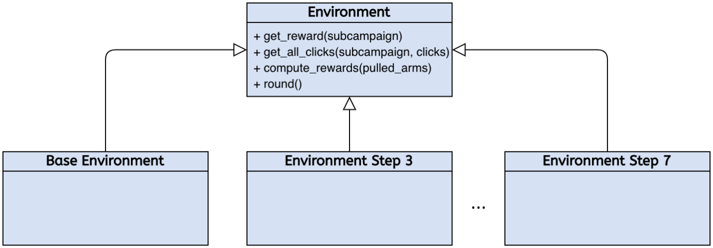

Environment
+ get_reward(subcampaign)
+ get_all_clicks(subcampaign, clicks)
+ compute_rewards(pulled_arms)
+ round()
Base Environment
Environment Step 3
...
Environment Step 7

Figure: Environment implementation in our solution.

Our solution (2)

We did the same as concerns for the subcampaigns. In particular, for representing a subcampaign we built a father Subcampaign class which is a generalization of all the subcampaigns used in the different steps. As before, this is an high-level abstraction of what we realized

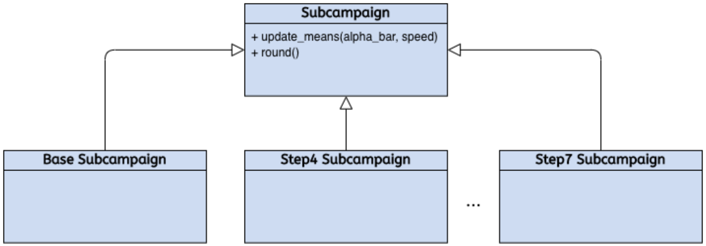


Figure: Subcampaign implementation in our solution.

Optimization Algorithm

As a company, obviously, we want to get the highest payoff from our advertising campaigns.

In order to optimize the profit, we adopted a Dynamic Programming algorithm assuming all the parameters to be known. Moreover, for the sake of simplicity, we assumed that the automatic bidding feature provided by the platform is used and therefore no bidding optimization needs to be performed.

The goal is to **find the best allocation of the budget**, that is to find the best way to spend the budget, for every feasible value of the budget.

Optimization Algorithm Formal Model (1)

To do that, we based our algorithm on a variation of the well-known multi-choice knapsack problem

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

s.t. : $\sum_{j=1}^{N} y_{j,t} \leq \bar{y}_t, \forall t \in T$

$\underline{y}_{j,y} \leq y_{j,t} \leq \bar{y}_{j,t}, \forall j \in N, \forall t \in T$

Optimization Algorithm Formal Model (2)

where

- N: number of subcampaigns (in our case, $N = 5$)
- $C = \{C_1, ..., C_N\}$ advertising campaign
- T: time horizon
- $t \in T$: instant of time
- $y_{j,t}$: daily budget of subcampaign C_j at time t
- v_j: value per click (impression) of subcampaign C_j
- $n_j(y_j)$: number of clicks (impressions) of subcampaign C_j given the values of budget y_j
- $\bar{y}_j$: cumulative daily budget constraint
- $\left[\underline{y}_{j,t}, \bar{y}_{j,t}\right]$: constraint for the budget of subcampaign C_j at time t

Optimization Algorithm Illustration (1)

At an high level, we can see the algorithm as a black-box that receives as input the expected revenues (based on the budget) and provides out the best way to allocate the budget, that is the 'best' (sub)budget to be spent for each subcampaign.

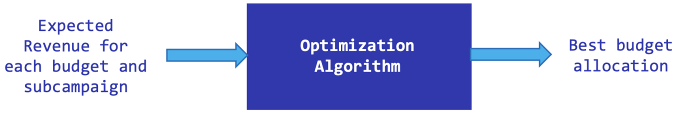


Figure: High-Level of abstraction for the Optimization Algorithm.

Optimization Algorithm Illustration (3)

Let's consider, for instance, that our maximum budget for the advertisement campaign is $B = 8'000€$ and that the table we have in input is:

	Budget (K€)								
Subcampaign	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
C1	0	90	100	105	110	0	0	0	0
C2	0	82	90	92	0	0	0	0	0
C3	0	80	83	85	86	0	0	0	0
C4	0	90	110	115	118	120	0	0	0
C5	0	111	130	138	142	148	155	0	0

Optimization Algorithm Illustration (4)

The algorithm provides us the following best allocation:

$$[2'000,\ 1'000,\ 1'000,\ 2'000,\ 2'000]\ \text{€}$$

and, as a result, an expected revenue of 502′000€. This means that if we want to maximize our profit, we should spent two thousands euros in subcampaign C1, C4 and C5 and half of this amount on subcampaign C2 and C3.

Of course, as expected, the sum of the budgets allocated is equal or less than B.

Optimization Algorithm Complexity Analysis (1)

Theoretically, the algorithm reported has a time-complexity of

$$O(N \cdot K^2)$$

where N is the number of subcampaigns, whilst K is the number of values for the daily budget.

We wanted to prove that our implementation was inline with this theoretical complexity. Thus, we tested this and the results are reported in the next slides.

Optimization Algorithm Complexity Analysis (2)

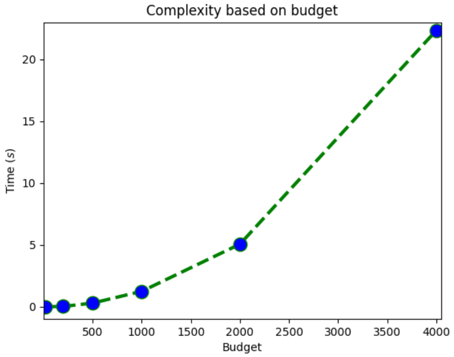


Figure: Complexity of the Optimization Algorithm. As we can see, the time is quadratic with respect to the number of values of the daily budget.

Optimization Algorithm Complexity Analysis (3)

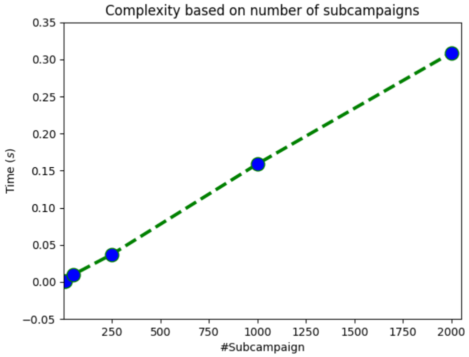


Figure: Complexity of the Optimization Algorithm. As expected, the time is linear with respect to the number of subcampaigns.

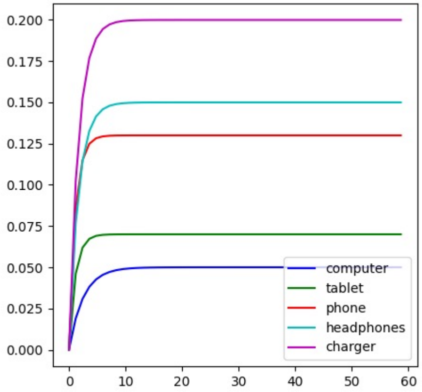
Optimization with uncertain α-functions

We now focus on the situation in which the binary features cannot be observed and therefore data are aggregated. In this setting, we do not have the α-functions but we need to learn them.

To do that, we designed a **combinatorial Gaussian Process bandit** algorithm, that is the composition of bandit algorithms and Gaussian Process.

Gaussian Processes (1)

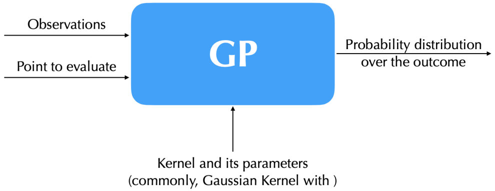
An example of the function we want the GP to learn is the following one (corresponding to a given subcampaign):



Gaussian Processes (2)

The idea is that we can assume there is some correlation among the points since doing so let us avoid to discretise the curve and have a too large number of samples to deal with it.

The idea of the GP is the following: it take as input the observations we have collected so far and the point to estimate and provides out the probability distribution over the outcome. To run the algorithm we also need to define a Kernel (commonly Gaussian) and its parameters.


Observations
Point to evaluate
GP
Probability distribution over the outcome
Kernel and its parameters (commonly, Gaussian Kernel with)

Gaussian Processes (3)

In our case, we have that

- **Observations**: collection of $(budget, click)$ points
- **Point to evaluate**: a $(budget, click)$ point
- **Kernel and its parameters**: a Gaussian kernel

$$\begin{aligned} k((budget, click), (budget', click')) &= k(\mathrm{x}, \mathrm{x}') \\ &= \exp\left(\frac{-||\mathrm{x} - \mathrm{x}'||^2}{2l^2}\right) \end{aligned}$$

where we posed $l = 1.0$.

Combinatorial Bandit Algorithms

As said, the arms (budgets) are somehow correlated and the reward of a budget provides some insights about the rewards of its near arms.

A combinatorial GP bandit algorithm exactly does what a standard GP bandit does, except that it allows to pull any set of arms satisfying some combinatorial constraint.

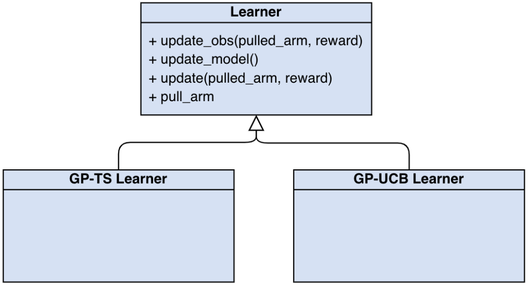
For us, the constraint to be satisfied is defined by the multi-choice knapsack constraint we have seen before. As a matter of fact, it strict the cumulative budget not to overcome the maximum capital budget B.

Our Solution

We build two learners for implementing what we explained so far:

- A **Gaussian Process - Thompson Sampling (GP-TS) learner**
- A **Gaussian Process - UCB (GP-UCB) learner**

As we did for the subcampaigns and the environment, we build a generic **Learner** class which generalize all the other learners


Learner
+ update_obs(pulled_arm, reward)
+ update_model()
+ update(pulled_arm, reward)
+ pull_arm
GP-TS Learner
GP-UCB Learner

GP-TS Algorithm

At every time $t \in T$

1. For every subcampaign $j \in N$, for every arm $a \in A$:

$$\bar{\text{r}}_{j,a} \leftarrow \text{learner}\,[\,j\,]\,.\mathit{sample}(a)$$

2. Execute the optimization algorithm

$$\mathbf{SA} \leftarrow \mathit{optimization}(\bar{\mathbf{r}})$$

   Where:

   - $\bar{\text{r}}_{j,a}$ is the expected reward of sub-campaign j and arm a
   - $\bar{\mathbf{r}} \in \mathbb{R}^{N \times |A|}$ is the matrix of the expected rewards
   - $\mathbf{SA} \in \mathbb{N}^N$ is the best super arm according to the optimization algorithm (i.e. the arms that must be pulled in order to get the maximum reward w.r.t. the functions learned so far)

3. For every subcampaign $j \in N$, play arm $\mathbf{SA}_j$
4. Update the GPs according to the observed rewards so far

GP-UCB Algorithm

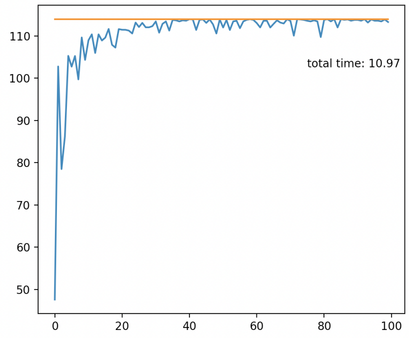
The only difference w.r.t GPTS is that in GP-UCB we also consider the upper bound $B_{a,t}$ that takes into account how much an arm have been played. This term, multiplied by the variance provided by the Gaussian Process σ gives to the algorithm a higher exploration approach. At every time $t \in T$

1. For every subcampaign $j \in N$, for every arm $a \in A$:

$$\bar{\mathrm{r}}_{j,a} \leftarrow \text{learner}\,[\,j\,]\,.sample(a) + \sigma \cdot \sqrt{\frac{2log(t)}{n_a(t-1)}}$$

2. Execute steps 2-4 of GPTS

GP-TS Algorithm results



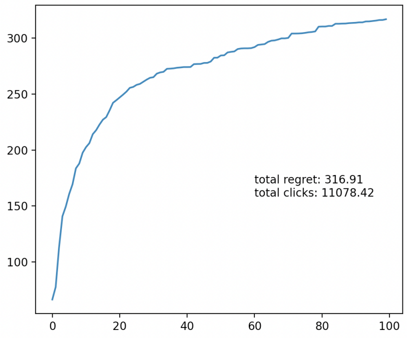
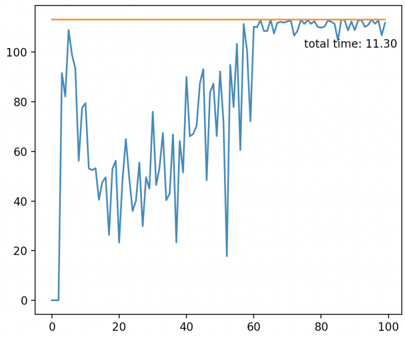


Figure: On the left, we can see the learner is able to reach the clairvoyant solution (in orange) just after roughly 30-time steps. On the right, we can see the regret that tends to flatten over time.

GP-UCB Algorithm results



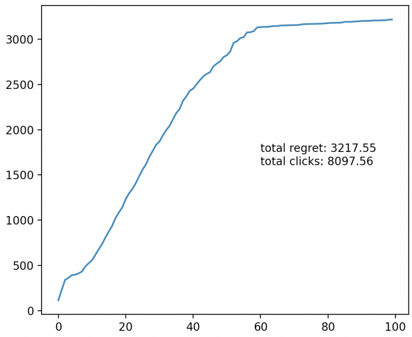


Figure: We can see the learner takes more time to learn with respect to the previous implementation. The reason may be that UCB tries to balance out exploitation and exploration and "lose" time doing so (even though the exploration is already handled by the GP).

GP-UCB and GP-TS Comparison (1)

Although once convergence is reached the two algorithms tend to be equally performant, we can see that GP-TS seems to be faster than GP-UCB.

On the other side, though, we have that UCB result is more reliable than TS, since is less sensitive to initial values.

GP-UCB and GP-TS Comparison (2)

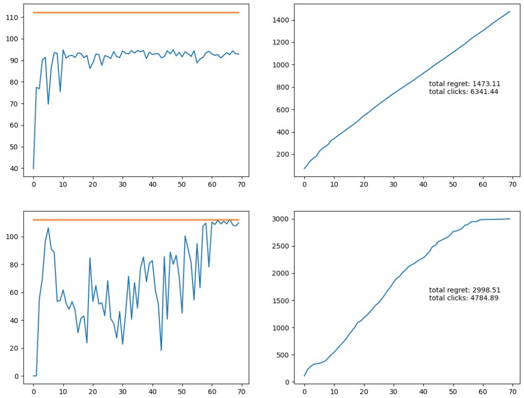


Figure: On top GP-TS, whilst on bottom GP-UCB.

GP-UCB and GP-TS Comparison (3)

In this example, we can see the GP-TS (figures on top) reaches convergence far from optimal.

This happens because if on the first samples, the arms of a sub-campaign are given 0-values, the Gaussian Process will define an all zero function that causes the optimization algorithm to never choose its values.

GP-TS Empirical Regret and Upper Bounds

For the combinatorial GP-TS, from theory, we have

$$R_T \leq \sqrt{\frac{2\Lambda^2 \cdot N \cdot T \cdot B \sum_{k=1}^{N} \gamma_{k,T}}{\log\left(1 + \frac{1}{\sigma^2}\right)}}$$

with probability $1 - \delta$, where $B = 8 \log\left(2 \frac{T^2 MC}{\delta}\right)$. For us

- $T = 100$ (horizon)
- $N = 5$ (number of GP learners, i.e., number of subcampaigns)
- $\delta = 0.95$
- $B = 20$ (number of arms)
- $\sigma^2 = 0.5$ (maximum variance of the GP-TS)
- $\gamma_{k,T} = 1.0$ (information gain $\forall k$)
- $\Lambda = 0.75$ (constant of the problem)

GP-TS Empirical Regret and Upper Bounds

We have computed the upper bound and we obtained that

$$R_T \leq 547$$

We run the algorithm for many times and we obtain an average ratio of

$$AverageRatio = \frac{\text{Mean Empiric Regret}}{547} = \frac{314}{547} \approx 56\%$$

Optimization with uncertain α-functions and #items sold

Now, we want to do the same as we did before, but considering the fact that also the number of items sold per product are uncertain.

Changes involve mainly the environment, whereas for the learners we're going to increase the alpha parameter of the GPs in order to avoid over-fitting.

Our Solution (1)

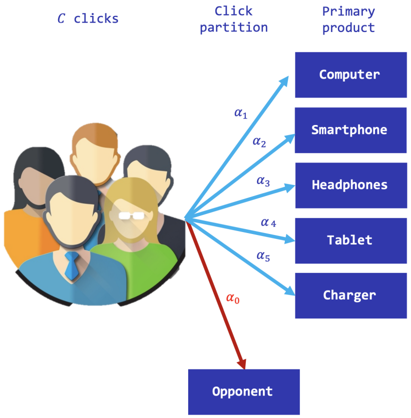
To model the uncertainty about the items sold we exploited the Dirichlet Distribution. As a matter of fact, as we explained in the intro, every day some customers lands on our web-site and the rest visits a competitor web-site.

The Dirichlet is well-suited for this task. In our scenario, will generate a vector of 6-elements

$$\alpha = \left[\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\right]$$

where $\sum_{i=0}^{N+1} \alpha_i = 1$, α_i is the ratio of users that land on web-page where the primary product P_i is shown ($i = 1, ..., 5$) and α_0 is the ratio of users will land on a competitor's web-page.

Our Solution (2)



We assumed that the users that will land on our website are those who will buy an item, while those that go to a competitor web-site are those that do not buy anything.

Our Solution (3)

For instance, in a generic day d, suppose the ratio of the users is the following one

$$\alpha_d = \left[0.3, 0.1, 0.2, 0.1, 0.1, 0.2\right]$$

This means that, on day d (assuming to have $C = 1000$ clicks)

- $P_1 \quad \rightarrow \quad C \cdot \alpha_1 = 100$ items sold
- $P_2 \quad \rightarrow \quad C \cdot \alpha_2 = 200$ items sold
- ...
- 300 clicks have not been converted into items sold

Results (1)

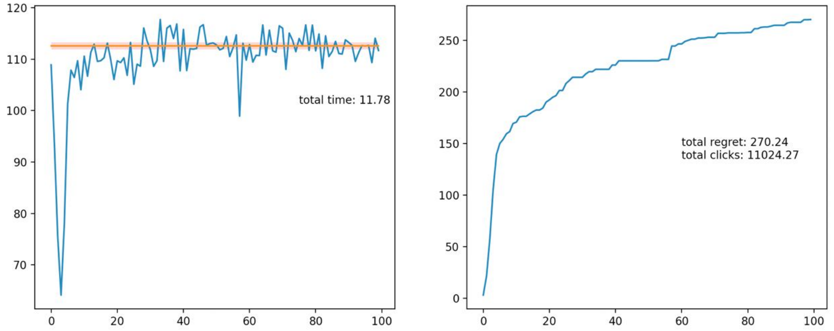


Figure: Optimization algorithm with uncertain α-functions and number of items sold.

Results (2)

In the left graph we can see that the algorithm seems to perform even better than the clairvoyant. This is actually impossible w.r.t the definition of clairvoyant. Indeed, this happens since is not possible, in our scenario, to run the clairvoyant algorithm on the same data of the learners. This because we can't reproduce the variance of the learner's reward playing a different arm (the clairvoyants).

The regret is considered to be 0 if the reward of the learner is equal or higher to the lower bound of the clairvoyant. Doing this we get a more reliable graph (otherwise we could get a non monotone function as regret)

Optimization with uncertain graph-weights

At the very beginning we showed the relation graph and we assumed the weights were known. Now, we remove this assumption.

The goal is to build an algorithm that actually does the same as before but indeed with uncertain graph weights.

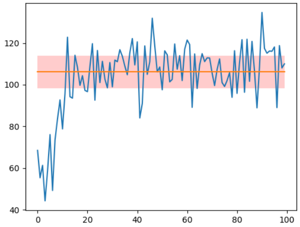
Our solution

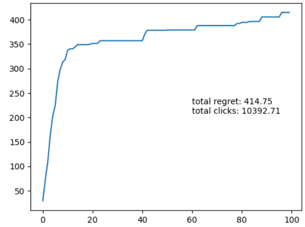
To encode the uncertainty about the graph-weights, we added a non-null variance to the adjacent matrix that describe the relation graph.

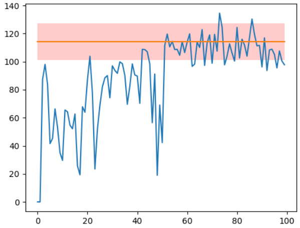
This way, we get the expected value plus the effect of a noise that represent the uncertainty on the weights.

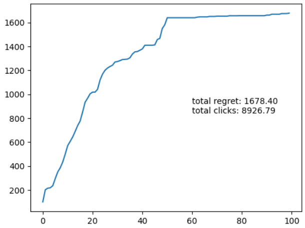
Having uncertain graph weights means, taking again the example did at the very beginning, that the probability a user will buy a computer along with a charger is not necessarily equal to 0.9 but we will say it is $0.9 \pm \varepsilon$ with $\varepsilon > 0$.

Results (1)


120
100
80
60
40
0
20
40
60
80
100


400
350
300
250
200
150
100
50
0
20
40
60
80
100
total regret: 414.75
total clicks: 10392.71


140
120
100
80
60
40
20
0
0
20
40
60
80
100


1600
1400
1200
1000
800
600
400
200
0
20
40
60
80
100
total regret: 1678.40
total clicks: 8926.79

Results (2)

We can clearly see that in this case the the variance is much higher for all the algorithms (also for the clairvoyant). This is due to the fact that as said the connection between items is not exact but can have some variance

Abrupt changes

Now we assume that the demand curves could be subjected to some abrupt changes. We're going to use UCB-like algorithms in two different settings:

- A sliding window setting where the algorithms keeps as valid only the last k samples where k is the window size. We expect this algorithm to play worse than previous in a stationary environment, but better in the long term if there are few changes
- A change detection algorithm that learns for the first k samples and then checks if there are substantial cumulative changes in the rewards. If this happens it will reset the data related to that sub-campaign and arm and learns it again for k samples

Sliding Window

In this setting we use the same UCB algorithm used for the previous steps, but we take into account only the last k samples. This allows the algorithm to keep track of small changes on the environment.

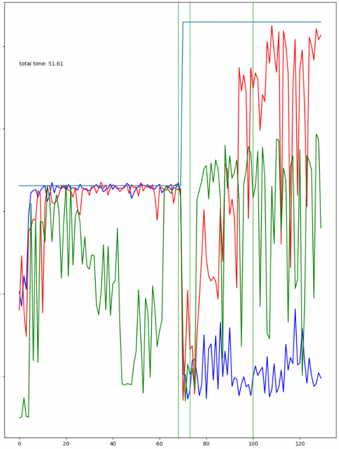
This algorithm doesn't work well with small window sizes and abrupt changes, indeed we will see that the regret is very high.

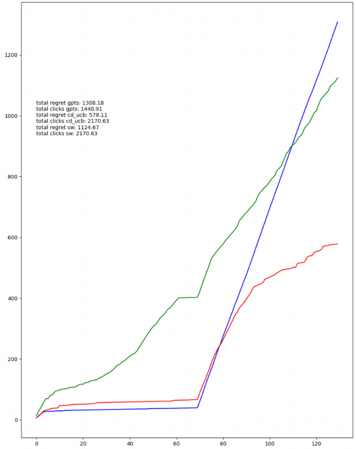
Change Detection

An algorithm that instead plays well in this environment is the CUSUM change detection. In this case we consider every small positive and negative variation w.r.t a mean computed on the first k rewards.

If the sum of all positive or negative variations exceeds a defined bound the algorithm clears the mean, and the algorithms starts the learning phase again.

Sliding Window, Change Detection & GP-TS Comparison


total time: 51.61


```
total regret gpts: 1308.18
total clicks gpts: 1440.91
total regret cd_ucb: 578.11
total clicks cd_ucb: 2170.63
total regret sw: 1124.67
total clicks sw: 2170.63
```

Sliding Window, Change Detection & GPTS

As we can see from the graph the CD (Change Detection) algorithm is the one that adapts better to the changes. On the other side GP-TS struggles to learn the change. Finally SW algorithm usually reaches a regret between the other two