Online Learning Applications

Advertising Project

Dorian Boille Raul Singh Roberto Reggiani
Davide Rigamonti Ole Martin Ødevald Ruud

Politecnico di Milano

December 2022

Presentation Outline

1 Introduction

2 Environment and Simulation

3 Optimization Algorithm

4 Uncertain α-functions

5 Uncertain α-functions and number of items sold

6 Uncertain Graph Weights

7 Non-stationary demand curve

8 Context generation

POLITECNICO
MILANO 1863

Project Scope

Assignment

The focus of this project is to optimize the budget allocated for the **advertisement campaigns** of 5 different items in order to maximize the profit of an *e-commerce website* that sells products to the public.

Each page has a **primary product** and two **secondary products**, each user that lands on a page has a possibility to buy the primary product and/or proceed to one of the secondaries with a given probability, the process repeats.

General hypoteses

The main general hypoteses that were given to us are:

- For every *primary product*, the *secondary products* to display and their order is fixed.
- The price of every product is fixed and it is equal to the margin.
- By clicking on a specific ad, the user lands on the corresponding *primary product*.
- No bidding optimization needs to be performed.

Learner interface

Description

In order for our project to have a more general outline, we decided to model a **generic learner interface** to standardize how our agents should be expected to behave while learning the budget distribution for a set of subcampaigns in a specific scenario defined by the environment.

In particular, each learner is characterized by the *learn* and *predict* actions. Every different type of learner will also receive a customized set of information filtered by the *masked environment* (in line with each project step) and potentially employs different algorithms to learn and predict the various budgets.

Learner interface

Code

```python
class Learner(ABC):

  # Updates the learner's properties according to the reward received.
  @abstractmethod
  def learn(self, interactions: List[Interaction], reward: float,
            prediction: np.ndarray):
    pass

  # Makes an inference about the values of the budgets for the subcampaigns
  # from the information got over time and the current environment
  @abstractmethod
  def predict(self, data: MaskedEnvironmentData) ->
              Tuple[np.ndarray, Optional[List[List[Feature]]]]:
    pass

  # Creates a figure and plots showing the status of learning progress
  @abstractmethod
  def show_progress(self, fig: plt.Figure):
    pass
```

Comparing results

For each different learner, when possible, we will show our results through the comparison of the rewards obtained by each algorithm against the rewards achieved (in the same enviroment setting) from the **Clairvoyant learner** and the Stupid learner. This, aside from theoretical formulations, will help us to define **upper bounds** and **lower bounds** for different solutions.

The *Stupid Learner* always subdivides the budget *equally* between products.

In particular, the **Clairvoyant learner** makes its "prediction" with full knowledge of the problem by identifying the **optimal superarm** (which corresponds to the optimal assigment of budgets) and placing it in a deterministic environment to collect an estimate of the best rewards achievable.

Comparing results

Clairvoyant learner

To estimate the **optimal superarm** we compute the **reward multipliers** (how much profit a single product brings on average) through hypotetical deterministic graph walks, performed by group of users following the expected values of the distributions defined on the graph edges. In addition, to remove any **non-determinism**, α-values are applied directly without any *Dirichlet noise* applied to them.

Given this definition of **clairvoyant reward** it may happen that, during simulation runs, the *learner reward* surpasses the *clairvoyant reward*; this is an expected and negligible behavior caused by the absence of noise in the clairvoyant formulation.

Comparing results

Regret

In order to evaluate the performance of our learners against the *clairvoyant result* we use the **cumulated regret** formulation:

$$R_T(L) := T\mu_{a^*} - \sum_{t=1}^{T} \mu_{a_t}$$

Usually we consider averaged results between multiple runs in order to reduce variance in the outcomes.

Gaussian processes

Gaussian Processes (**GPs**) are a powerful tool to tackle regression problems: by defining a **kernel** with its parameters and feeding the GP with the arms pulled by the bandit and their rewards it's possible to obtain a probability distribution over the outcome in form of arrays of **means** and **sigmas**. Different bandit algorithms use those values in different ways.

Each **GP** has been instantiated with the kernel: $Ck(\theta) * RBF(\ell)$ where *Ck* is the θ-constant kernel and *RBF* is the *Radial Basis Function* kernel of length ℓ, defined as:

$$exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right)$$

Gaussian processes

Regret

$$R_T \leq \sqrt{\frac{2\Lambda^2}{log(1+\frac{1}{\sigma^2})} CBT \sum_{k=1}^{C} \gamma_{k,T}} \text{ with probability of } 1-\delta$$

where:

$$B = 8log\left(2\frac{T^2MC}{\delta}\right)$$

C=5 is *number of subcampaigns*

M=20 is the *number of arms*

δ=5 is the *confidence* of the regret bound

σ=5 is the *maximum variance* of the GP

$\gamma_{k,T}$ is the *information gain* at time T for subcampaign k

Λ is the *Lipschitz constant* of the problem

MAB algorithms

TS and UCB

To tackle most of the problems in this project we will use two main **MAB** algorithms called **Thompson sampling** (**TS**) and **Upper Confidence Bound 1** (**UCB1**).

In the **UCB1** algorithm, every *arm* is associated with an **upper confidence bound** which provides an *estimation* of the reward gained by playing that specific arm, at every trial, the arm with the **highest upper confidence bound** is pulled and the bound is updated accordingly.

Some aspects of **TS** are similar to **UCB1** but the main difference is that in **TS** every arm is associated to a prior β distribution and every arm has a prior on its **expected value** based on it; at every trial the algorithm chooses the arm with the **best sample**, then, it updates the distribution of the chosen arm according to the observed realization.

2 Environment and Simulation

- Overview
- Environment structure
- Randomness in the Environment
- Simulation

Assumptions

E-commerce website

For this project, we are required to design an **Environment** that satisfies various constraints both on the e-commerce site's properties and on the users' behavior; in addition, since most of the tasks were generic, we had to come up with some of our own assumptions.

In particular we want to underline the following assumptions for the e-commerce website:

- The website has unlimited stock for the 5 different items.
- Actions on the various webpages are **perfectly observable** by the e-commerce website.

Assumptions

Users

...and we assume that the users present the following behaviors:

- Every day, there is a random number (subject to noise) of potential new users.
- The users can activate **parallel paths** while on the website.
- The number of items that a user will buy is a random variable, independent from any other variable; but every user that lands on the page of a product will buy at least 1 unit of it if it's under their **reservation price**.
- The behavior of users can be modelled as a graph where *nodes* represent product pages and *weights* represent the probabilities for a user to click from the primary item of the page to one of the secondaries.

Environment

Structure

The environment is modelled as a python dataclass containing the following attributes:

```
# The total budget to subdivide
total_budget: int

# Probability of every class to show up. They must add up to 1
class_ratios: List[float]

# Features associated to every class
class_features: Dict[Tuple[Feature, Feature], int]

# Price of the 5 products
product_prices: List[float]

# List of class parameters for each class and product,
# implemented as list of lists of UserClassParameters.
# Each class has distinct parameters for every product
classes_parameters: List[List[UserClassParameters]]
```

Environment

Structure

```
# Lambda parameter, which is the probability of osserving the
# next secondary product according to the project's assignment
lam: float

# Max number of items a customer can buy of a certain product.
# The number of bought items is determined randomly with
# max_items as upper bound
max_items: int

# Products graph's matrix. It's a empty matrix, should be
# initialized with populate_graphs
graph: np.ndarray

# List that constains for every i+1 product the secondary i+1
# products that will be shown in the first and second slot
next_products: List[Tuple[int, int]]

# Controls randomness of the environment
random_noise: float
```

Environment

Masked Environment

Alongside the **Environment** we define a **Masked Environment** with the purpose of *hiding crucial information* to the learners since each type of learner should only have access to a subset of all the information available in the environment dictated by the type of learner.

The masked environment isn't strictly needed in the project since the learners could easily ignore the extra information, however, we wanted to face the problem with an approach aimed towards reusability and extendability and in this case (as in many others down the line) we opted for a more **generalizable** solution.

Users

User parameters

We modelled the α-functions, which represent the ratio of population that may interact with a product given a fixed budget, as **exponential functions**.

In particular their *upper bound* represents the maximum expected number of interactions possible while the *maximum useful budget* is the amount of budget after which any budget increase would not lead to a ratio increase.

Users

User classes

Users are subdivided in classes based on their *2 binary features* for a total of *3 different classes*.

In particular, each user class is defined by its α-functions (one for each product plus the one for the non-strategic competitor) which define the probability of landing on a given product page.

Each α-function is defined by the values: **reservation price**, **upper bound** and **maximum useful budget**.

Randomness in the Environment

Non determinism

For the sake of representing a real scenario, most of the values that are not known a priori are *randomly generated* and every variable that evolves through time without our direct control has elements of randomness to it (for instance, each day we randomly get the number of active total users in our scenario by using a gaussian distribution with tunable mean and standard deviation).

Even though most of the randomness is tunable and controlled through seeded generators, there are still **impactful elements of non determinism** (i.e. the Dirichlet distribution) that are not possible to control in any way.

Daily Simulation

Purpose

The **Simulation** class is the main engine that brings together *learners* and *environment* by making them interact with each other while offering an interface to customize the execution.

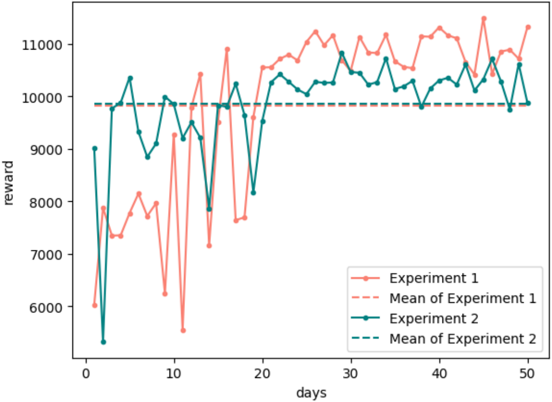
The basic idea of the simulation is to simulate a real scenario day by day using the environment to generate interactions with the website according to the budgets that the current learner proposed and then, feed the results back to the learner to make it actually learn.

Repeating the simulation execution step for each day until an arbitrary **time horizon** is reached grants us all the data needed to evaluate the performance of our learner.

Daily Simulation

Example

Simulation runs of two learners in an environment with total_budget = 400 and population_mean = 1000



POLITECNICO MILANO 1863

3 Optimization Algorithm

- General Problem
- Algorithm

Problem formulation

The purpose of the **optimization algorithm** is to find the *optimal budget* for each campaign in order to maximize the *profit*, which is defined as the difference between the profits gained from selling products and the advertisement expenses.

Basically, it is a maximization problem subject to an obvious constraint: the sum of the budget dedicated to each subcampaign can't be greater then the overall budget.

The optimization problem can be expressed as follows:

$$F = \max_{x_i \in B} \sum_{i=0}^{n} \alpha_i(x_i) p_i - x_i \quad \text{s.t.} \quad \sum_{i=0}^{n} x_i \leq B$$

Code

Functioning

Given a **budget assignment problem** over the matrix c, which is assumed to be divided into n rows (campaigns) and m columns (budget fractions from 0 to B *maximum budget*).

We solve this problem via **DP** and we start by setting up **two extra tables** to store the *best values* and their *relative allocations*, then we iterate over each row and column finding the best index for each cell by taking into consideration the sum between the current and the previous row matching budgets; we store the best values and their indexes in the corresponding tables.

The optimal allocation lies at the end of the of the updated table, however, in order to obtain a final allocation which fully utilizes the whole budget we trace back the best values table **greedily** adding sub-optimal allocations until our budget runs out.

Code

Example

```
dp = [c[0]]
allocs = [np.arange(m)]

for i in range(1, n):
    next_dp_row = []
    next_alloc_row = []
    for j in range(m):
        prev_row_r = dp[i - 1][j::-1]
        curr_row = c[i][: j + 1]

        row_sum = prev_row_r + curr_row

        # The best index is the index which we should select for
        # the maximum profit, it will be what we store in
        # allocation, and we will also store the value of the
        # row_sum at best_index to dp[i][j]
        best_index = np.argmax(row_sum)

        next_alloc_row.append(best_index)
        next_dp_row.append(row_sum[best_index])

    allocs.append(np.array(next_alloc_row))
    dp.append(np.array(next_dp_row))

dp = np.array(dp)
allocs = np.array(allocs)
# ...
```

Code

Example

```python
# ...
best_dp_index = np.argmax(dp[-1])

# As we stored the allocation of what the dp would mean, we now need to
# trace back through the allocs array to create the final allocation
last_alloc = allocs[-1][best_dp_index]

final_allocs = [last_alloc]
remaining_budget = m - last_alloc

for i in range(n - 2, -1, -1):
        # Based on the remaining_budget, we take the max of the remaining
        # possible cumulative values that we have in the dynamic
        # programming table
        sub_dp = dp[i, :remaining_budget]
        sub_best_index = np.argmax(sub_dp) if len(sub_dp) > 0 else 0

        next_alloc = allocs[i][sub_best_index]
        final_allocs.append(next_alloc)

        remaining_budget = remaining_budget - next_alloc

# As we constructed the the final allocation from the back to front,
# we need to reverse it before returning it
return np.array(list(reversed(final_allocs)))
```

4 Uncertain α-functions

- Contextual hypotesis
- Algorithm
- Results

Scenario

We now assume that the binary features of the users cannot be observed and therefore data is considered as **aggregated**.

Since the features of the users are **not observable**, the α functions' shape for each class is unknown.

As a result, in our scenario the learner receives all the interactions minus the parameters of the α functions.

Solving the problem

By gathering the aggregated reward for each product we are able to utilize those coarse rewards to generate feedbacks for our **MABs** and therefore train them on the aggregated interactions for each day.

In particular we exploit **Gaussian Processes** in conjunction with **MAB** algorithms such as **Thompson Sampling** and **UCB1** to exploit the continuity between the different arms.

We instantiate a **GP-MAB** for each subcampaign and each **MAB** will have `n_budget_steps` number of arms.

Algorithms outline

GP TS

GPTS is a variant of **TS** implemented using **Gaussian Processes**:

- For each day t, we gather a sample from each arm a:

$$\tilde{\theta}_a \leftarrow \text{Sample}\left(\mathbb{P}(\mu_a = \theta_a)\right)$$

- Play arm a_t defined as:

$$a_t \leftarrow arg \max_{a \in A} \left\{ \tilde{\theta}_a \right\}$$

- Update the **Gaussian Process** with the reward obtained.

Algorithms outline

GP UCB1

GPUCB1 is a variant of **UCB1** that takes advantage of the **Gaussian Processes** confidence interval and models it as the confidence bound; apart from the arm choice, the learning process is equal to **GPTS**.

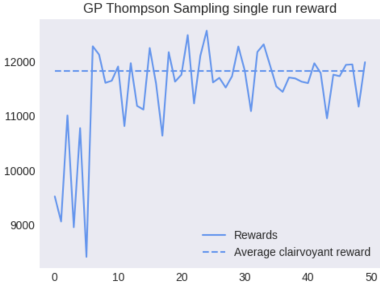
$$a_t \leftarrow arg \max_{a \in A} \{\mu_{t-1} + \delta \sigma_{t-1}\}$$

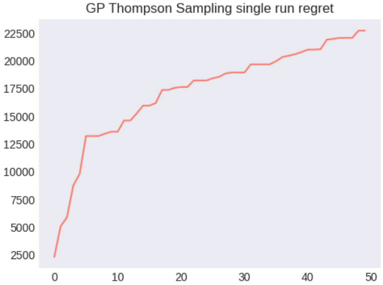
Code snippet that calculates the **upper bound**:

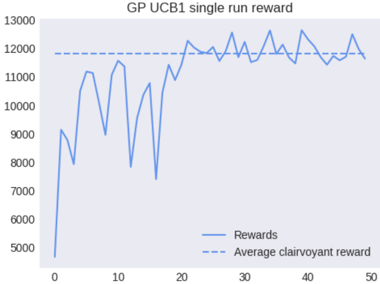
```
def estimation(self):
        upper_bounds = (self.means + self.confidence * 1.96 * self.sigmas)
                        * self.normalize_factor
        return upper_bounds
```

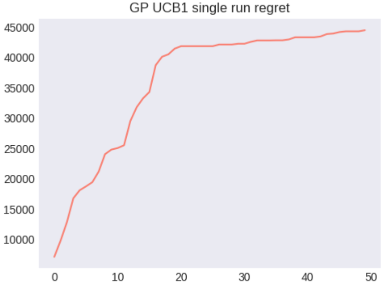
Single run reward and regret

Thompson Sampling and UCB


GP Thompson Sampling single run reward

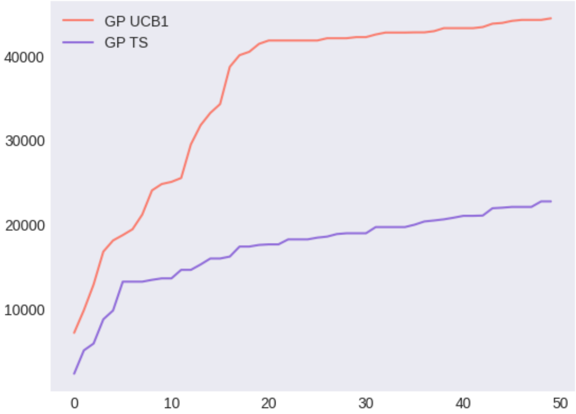

GP Thompson Sampling single run regret


GP UCB1 single run reward


GP UCB1 single run regret

Regret comparison

Thompson Sampling and UCB

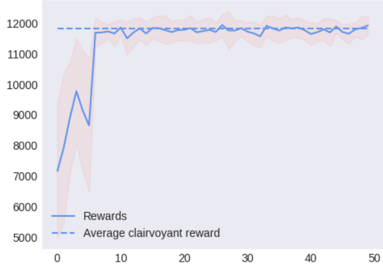


All tests are done using the `example_environment` default values, *population mean* of 1000, *variance* of 10 and 20 *budget steps*.

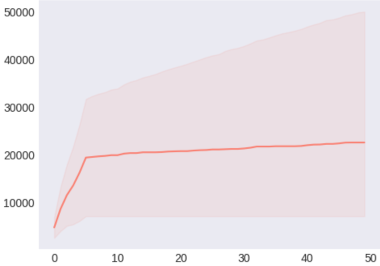
Average reward and regret

Thompson Sampling and UCB

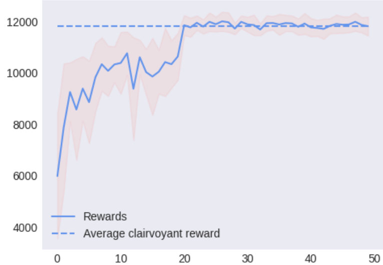
GP-TS average reward over 15 experiments



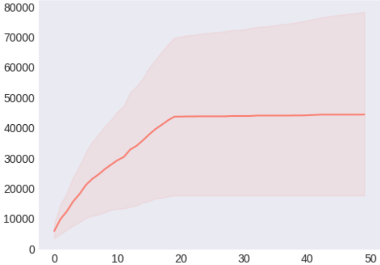
GP-TS average regret over 15 experiments



GP-UCB1 average reward over 15 experiments

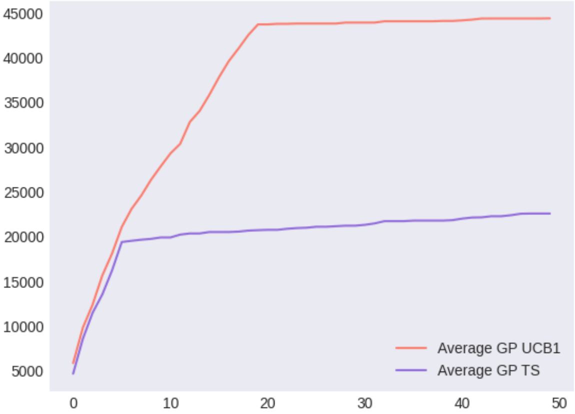


GP-UCB1 average regret over 15 experiments



Average regret comparison

Thompson Sampling and UCB



Results

Overall we can observe more instability in the **TS** algorithm, but a faster convergence w.r.t to the **UCB** approach.

Both algorithms clearly converge to the optimal solution at different rates while respecting a linear cumulative regret bound.

Average results over 15 runs at time horizon $T = 50$:

	Reward	Regret	Deviation
	μ	μ	σ
GPTS	11610.53	31024.67	440.36
GPUCB	11737.73	43686.67	349.47

5 Uncertain α-functions and number of items sold

- Contextual hypotesis
- Algorithm
- Results

POLITECNICO
MILANO 1863

Scenario

In this case the e-commerce website doesn't register neither the **units sold** for each product nor the **class parameters**.

Since this isn't a change that affects the enviroment directly, there will not be a separate **masked environment** mask to hide the *units sold* to the learner, therefore the extra information will be ignored by the learner.

We expect worse results overall since the learners are working with less data and therefore their prediction will have to factor in more **uncertainty**.

Algorithm

Algorithm outline

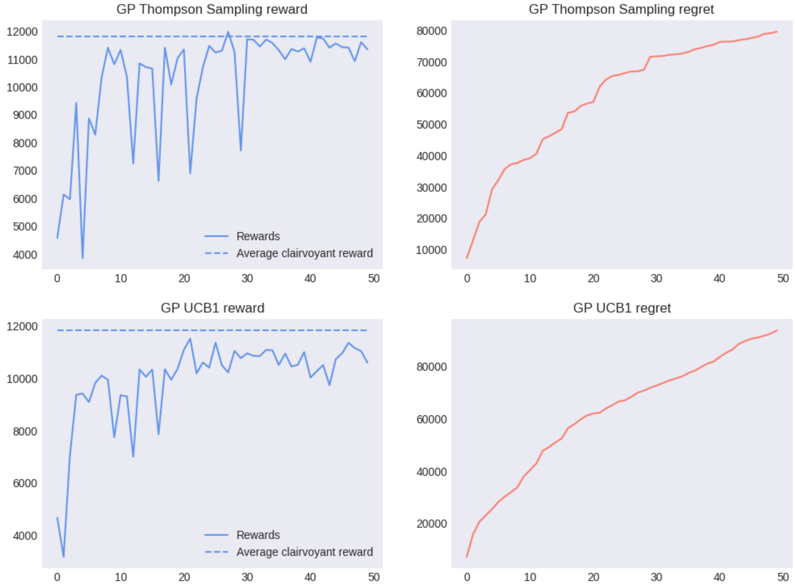
The learner that we modelled for this specific scenario bares a lot of *similarities* w.r.t. the learner used for the previous step as they both function following the same workflow.

However, in this case, the **learn** function is only allowed to gather information from the generic reward obtained for the day.

```
def learn(self, _, reward: float, prediction: np.ndarray):
        for i, p in enumerate(prediction):
                prediction_index = np.where(self.budget_steps == p)[0][0]
                self.product_mabs[i].update(prediction_index, reward)
```

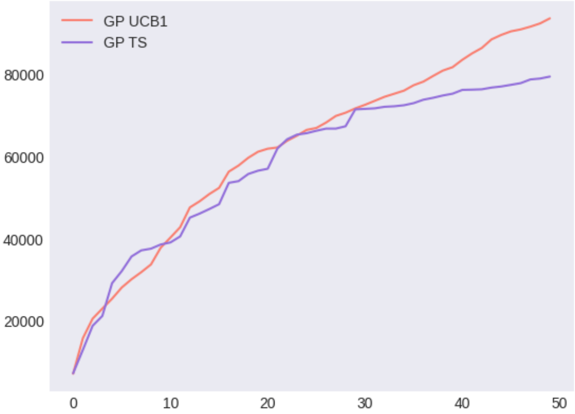
Single run reward and regret

Thompson Sampling and UCB



Regret comparison

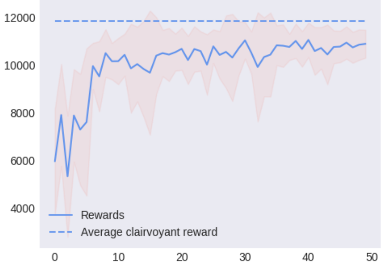
Thompson Sampling and UCB

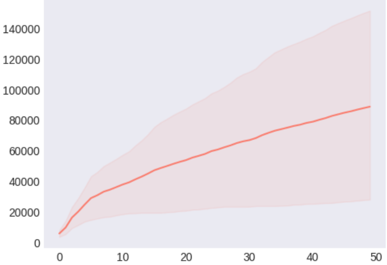


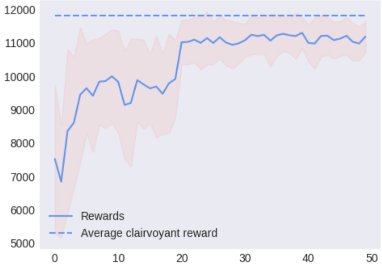
All tests are done using the `example_environment` default values, *population mean* of 1000, *variance* of 10 and 20 *budget steps*.

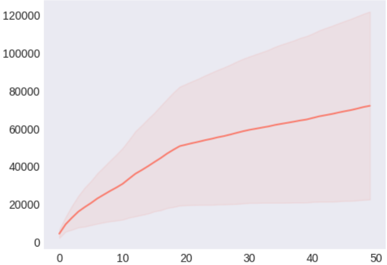
Average reward and regret

Thompson Sampling and UCB


GP-TS average reward over 15 experiments

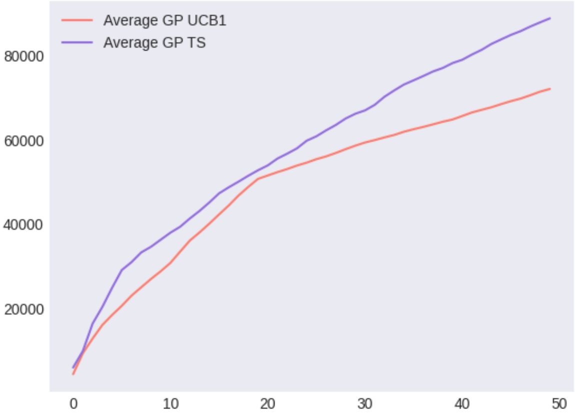

GP-TS average regret over 15 experiments


GP-UCB1 average reward over 15 experiments


GP-UCB1 average regret over 15 experiments

Average regret comparison

Thompson Sampling and UCB



Results

In this scenario, results are worse w.r.t. the previous step since the learners work with less information.

TS seems to reach the optimal solution but is much more unstable than the previous case, however, if we use the **UCB** approach we are not guaranteed to find the optimal arm as sometimes it settles on a suboptimal solution.

On average we can observe that **UCB** performs slightly better than **TS** probably due to a more unrealiable environment resulting in learners that are more "unsure" nullifying the advantage dictated by the randomness of the latter.

Results

Average results over 15 runs at time horizon $T = 50$:

	Reward	Regret	Deviation
	μ	μ	σ
GPTS	10772.60	97205.53	904.08
GPUCB	11029.20	72557.20	510.36

POLITECNICO
MILANO 1863

6 Uncertain Graph Weights

- Problem
- Implementation
- Results

Scenario

Hypotesis

In this scenario the only uncertain parameters are the **graph weights**.

Alongside the other operations, the learner will run a **graph estimation** algorithm to build an accurate representation of the graph at each time step.

Meanwhile, since the α-**functions** are known, there is no need to estimate them and they can be applied directly in the optimization problem.

Initial solution

In the first interpretation of this assignment we set out to exploit graph influencing techniques in order to tackle this problem, however, in the long run the results were **suboptimal** and we decided to opt for a different approach.

Nonetheless, the graph influence related functions **still exist** in the codebase for posterity reasons, those are: `get_influence_per_product`, `make_influence_graph` and `get_influence_of_seed`.

In the end we opted for a more *experimental approach* by simulating the hypotethic behavior of a fixed set of people walking the graph, following our *estimated weights* and observing which products resulted in more return; we were able to produce meaningful budget assigments and tune the estimated graph weights over time.

Solving the problem

Prediction

Since there is no need to estimate the α-functions, the best allocation is calculated using the `find_optimal_superarm` function used to obtain the best estimation when the α-functions are known.

The only difference is that a custom graph needs to be specified since we don't have access to the real graph weights.

Graph estimation

The learner contains a graph representation that is updated at each time step; the graph representation is not updated directly but through a function called `graph_estimate`, which collects samples from various **beta distributions** for each graph weight deleting *self-loops* and non-existent edges.

The parameters of the **beta distributions** (α and β) are defined for each edge of the graph and represent the effective quantities that are updated whenever the function `learn` is called on the learner.

Parameters update

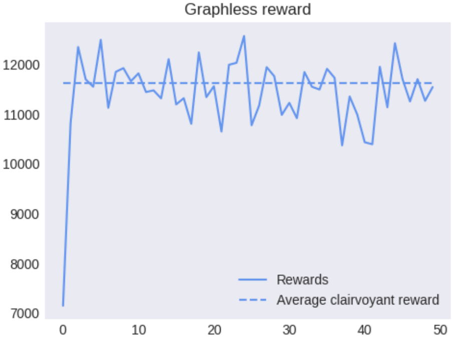
Whenever we want to make the **graphless learner** learn by calling its dedicated function, the learner analyzes all of the interactions given as a parameter and looks at the path that the users traversed by comparing it with the current graph representation.

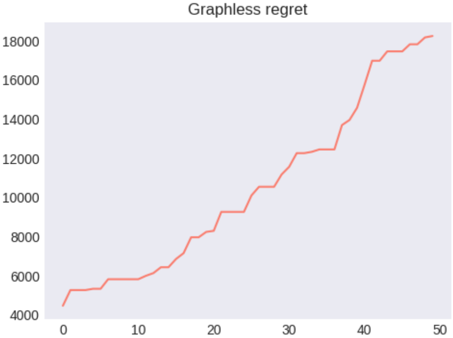
If a certain edge is taken or not by a given user, the α and β parameters are updated accordingly.

When `predict` is called again, the graph representation is then generated from scratch by drawing samples from the **beta distributions** with the updated parameters.

Reward and regret

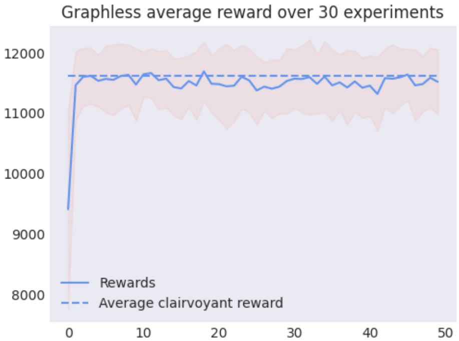
Graphless

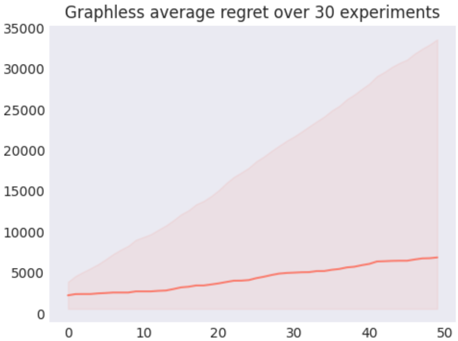




Average reward and regret

Graphless


Graphless average reward over 30 experiments


Graphless average regret over 30 experiments

All tests are done using the `example_environment` with custom values, *population mean* of 1000, *variance* of 10 and 20 *budget steps*.

Results

We can observe that the regret is exceptionally low w.r.t the previous learners, this is most likely due to the fact that the **graphless learner** holds most of the important information from the environment and is able to give accurate estimations from the start.

However, some variance is always present due to the *imperfect graph estimation* and *environment non-determinism*.

Average results over 30 runs at time horizon $T = 50$:

	Reward	Regret	Deviation
	μ	μ	σ
Graphless	11515.87	6845.87	534.97

POLITECNICO
MILANO 1863

7 Non-stationary demand curve

- Environment behavior
- Algorithms
- Results

Non-stationary behavior

Overview

By using the **non-stationarity** assumption we are taking into consideration that the demand curve could be subject to changes over time and isn't necessarily fixed as we have seen in other steps.

There are 2 main types of *non-stationary behaviors*:

- **Abrupt changes**: where it's possible to identify different phases with different phase-wise stationary demand curves.
- **Smooth changes**: where the demand curve changes over time in a continuous manner.

As per specifications, we will only consider **abrupt changes** in the scope of our project.

Non-stationary behavior

Abrupt changes

Abrupt changes are usually experienced when an important event strikes the market *(e.g. a new product that shifts the interests of the users is released or an historical event shapes the opionion of people)*; change isn't necessarily bad, however, it may impact negatively the prediction of learners that were created with a static environment in mind.

Neither **UCB** nor **TS** account for abrupt changes since it's almost impossible for them to try a superarm that was deemed as *unoptimal* over the past iterations (while it might have become optimal after an *abrupt change*), therefore we expect to see a significant **reward drop** from them after an *abrupt change*.

Approaches

Sliding window and Change detection

There are 2 main approaches to deal with a **non-stationary environment**

- **Sliding window**: only consider the last τ samples for predictions.
- **Change detection**: detect when a change has happened and adapt accordingly.

It's clear that **Sliding window** approaches are more fit for **smooth changes** while **Change detection** approaches work better with **abrupt changes**, however it's important to note that both approaches can be utilized to deal with any *non-stationary* scenario.

Sliding window

Sliding window for TS and UCB

- **SW-GPTS** only differs in the **gaussian process** update since the older samples are progressively removed from the estimation as time goes on.
- **SW-GPUCB** also differs in the confidence bound formulation and the new arm choice:

$$a_t = \begin{cases} a_{\overline{t}} & \text{if } \exists \overline{t} \mid n_{a_{\overline{t}}}(t-1,\tau) = 0 \\ arg\ \max_{a \in A} \left\{ \mu_{t-1,\tau} + \delta \sigma_{t-1,\tau} \right\} & \text{otherwise} \end{cases}$$

Change detection

We used a **reward-based algorithm** as a *change detection algorithm*, it works by comparing the last k rewards with the newest k rewards ($k = 4$ by default) and if the difference between their means exceedes a certain threshold ω ($\omega = 1000$ by default) the learner is **reset** and is therefore able to learn a new optimal superarm from scratch.

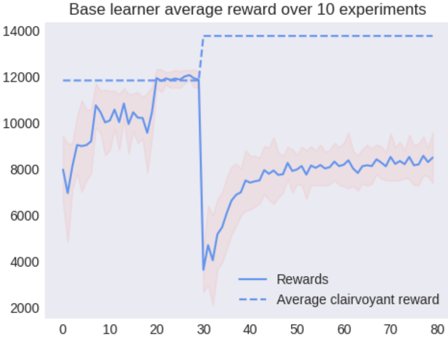
Formally:

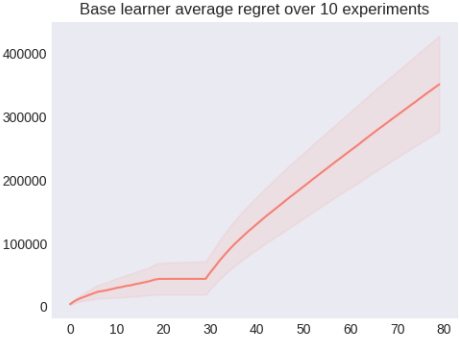
$$\text{RESET learner if } t \geq 2k \;\wedge\; \sum_{i=t-2k}^{t-k} r_i - \sum_{i=t-k+1}^{t} r_i > \omega$$

N.B. there is an offset of 1 in the representation of t between theory and code

Average reward and regret

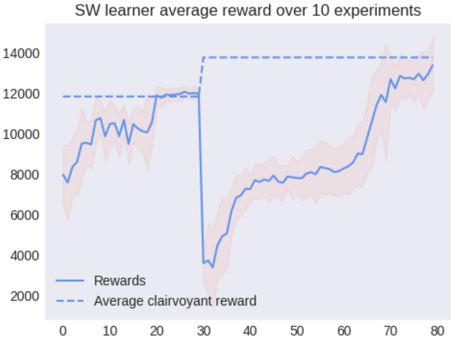
Base learner

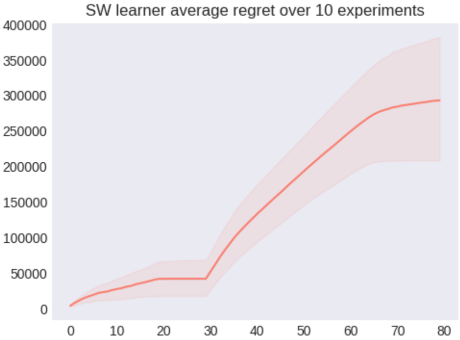




Average reward and regret

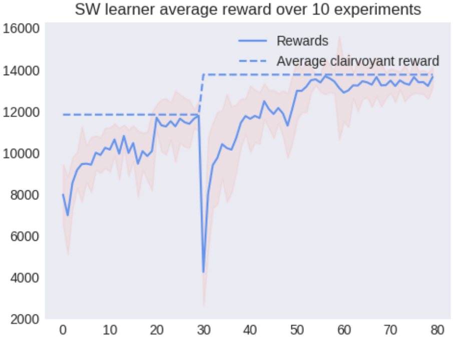
Sliding window

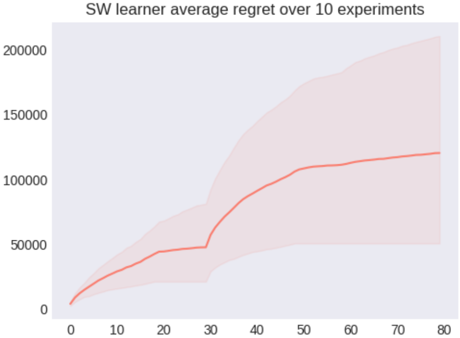

SW learner average reward over 10 experiments


SW learner average regret over 10 experiments

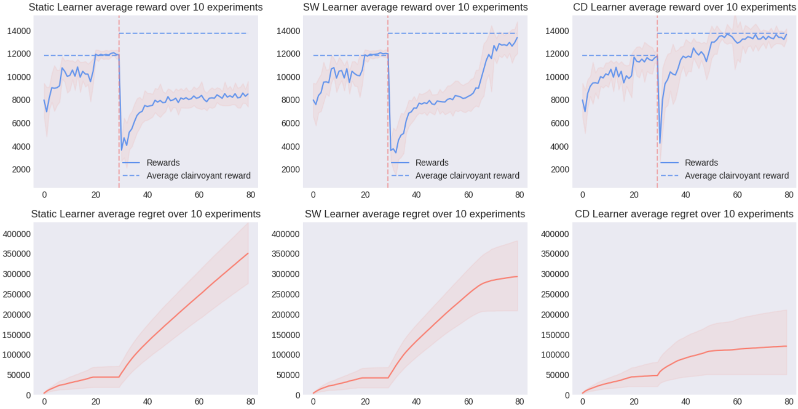
Average reward and regret

Change detection


SW learner average reward over 10 experiments


SW learner average regret over 10 experiments

Reward and regret comparison



All tests are done using the `example_environment` with default values, *population mean* of 1000, *variance* of 10 and 20 *budget steps*. Only **UCB** algorithms were evaluated. 1 breakpoint at 30 days.

Results

From the results it's quite clear that the **change detection** algorithm is by far the best one in this specific scenario, while it's interesting to see that the **sliding window** approach it's slower to stabilize (due to the fact that the samples from the old environment must completely exit the window to not be considered) and the **base learner** approach isn't able to adapt, as expected.

Generalizing, we can say that if the **number of breakpoints** m is small enough with respect to the **time horizon** T to the power of α we have that the **regret** is of the order $O\left(|A| T^{\frac{1+\alpha}{2}}\right)$.

Results

Average results over 10 runs at time horizon $T = 80$:

	Reward	Regret	Deviation
	μ	μ	σ
Base learner	7717.40	376073.10	877.73
Sliding window	12571.80	271383.60	908.26
Change detection	13452.30	126329.10	684.53

8 Context generation

- General Remarks
- Assumptions
- Implementation
- Results

Theory

Basics

The goal of **context generation** and **contextual bandit algorithms** is to employ a (partially) disaggregated approach in order to better exploit the differences of the users that belong to different classes.

This is made possible by using a specialized Learner for each context and an *offline context generation algorithm* that decides which contexts are worth to target by isolating them from the aggregated data. This algorithm is ran every two weeks.

Each context is able to target a set of features, and a **context structure** is a set of contexts that targets all the existing features without any overlap.

Implementation challenges

Weaknesses

Context generation has been, by far, the *toughest* task that we faced on this whole project.

Even after many meetings and discussions we felt that there was something that didn't click with our interpretation since we didn't find a way to give a complete meaning to the dataset collected by our active bandits due to how the prediction and the rewards worked in our scenario.

In the end, our final results for this step were **unsatisfactory** and there is still wide room for improvement.

Implementation assumptions

Compromises

Given the difficulties that we encountered, we decided to make some compromises in order to still carry out the task to an end:

- The e-commerce website company owns a simulation capable of simulating interactions.
- Learners are trained on the fake simulation since we can't get enough information from the logs.
- The best expected reward for a given context is considered as the maximum reward experienced by a learner on a fake simulation run.

Implementation assumptions

Potential problems

Given our interpretation, some quantities were particularly difficult to identify in a correct manner, especially the **expected value** μ **of the optimal arm** a^* **for a context** c (written as μ_c) that we resorted to calculate as the mean reward over a fake simulation on the context c.

Additionaly we used the **Hoeffding bound** to compute lower bounds for both the *context probability* and the *context expected value*.

These particular points are critical parts that absolutely need to be revisioned and most definetly are concurring causes to the unsatisfactory results.

Greedy algorithm

Part 1 of 2

We generate contexts following a **greedy algorithm** that, given a context c:

- Considers all the possible binary 1-feature splits $\{c_i^0, c_i^1\}_n$ for the features not present in c.
- Creates and evaluates a new learner for each split c_i^j, obtaining the **context probability** p_i^j and the **best expected reward** μ_i^j.

$\rightarrow$

Greedy algorithm

Part 2 of 2

→

- Evaluates the **splitting condition** on each couple of splits (c_i^0, c_i^1) w.r.t the original context c while deciding which split is the best one (if it exists).
- If a split has been made, repeat all the operations recursively on the new contexts until *no split is made* or *no split is possible*.

Context generation

Split condition

Since utilizing multiple contexts is an expensive operation and our *greedy algorithm* is **not** guranteed to find the optimal solution, we want to be sure that when we introuduce new contexts, it is actually worth to do so.

We use a particular *splitting condition* that exploits lower bounds in order to set a higher threshold for the quality of our splits:

$$\underline{p}_i^0\underline{\mu}_i^0 + \underline{p}_i^1\underline{\mu}_i^1 \geq \underline{\mu}_i$$

for context i defined by a binary feature split on 0 and 1.

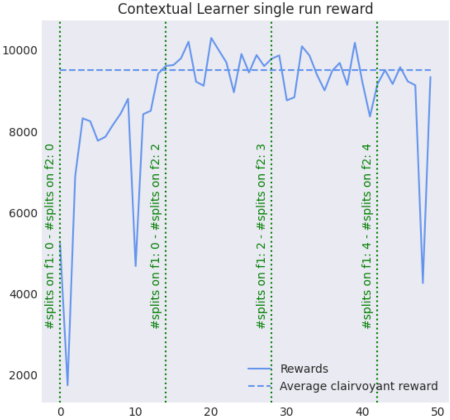
Context utilization

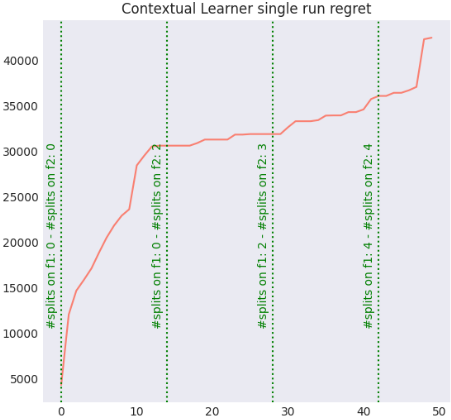
Once a certain number of contexts has been established, a **Contextual Learner** will be able to assign an **AlphaUnitless Learner** to each context and obtain their raw predictions by utilizing the function **predict_raw** (which returns an un-optimized array of predictions) and then running the optimization algorithm on all the predictions gathered this way.

The resulting *superarm* is then passed to the **Simulation** which, in return, generates the interactions for the day by weighting the different classes according to the disaggregated predictions.

Reward and regret

Contextual learner

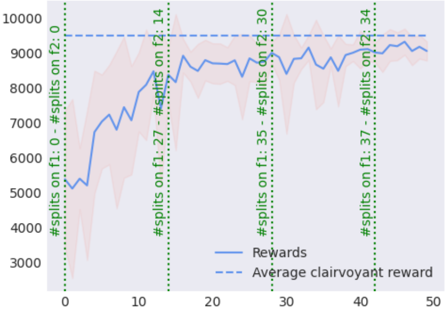




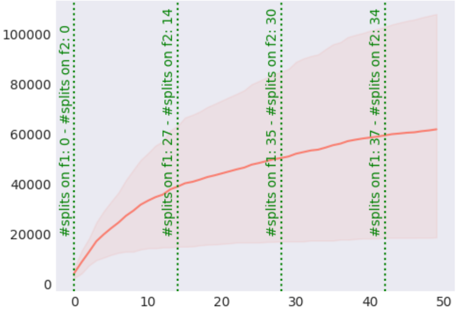
Average reward and regret

Contextual learner

Contextual Learner average reward over 10 experiments



Contextual Learner average regret over 10 experiments



All tests are done using the `example_environment` default values, *population mean* of 800, *variance* of 10 and 20 *budget steps*. Only **TS** algorithms were evaluated.

Results

Even though the results might not be completely satisfying as we would expect a performance superior w.r.t. the *clairvoyant estimation*, on average we still perform better than the **AlphaUnitLess learner** as we seem to converge to the optimal superarm for the aggregated contexts.

Overall the learners seem to prefer **early splitting** but various parameters can be tuned (e.g. bound confidence) to make the process of splitting on contexts more *rigorous*.

Average results over 30 runs at time horizon $T = 50$:

	Reward	Regret	Deviation
	μ	μ	σ
Contextual learner	9065.30	61719.2	273.67

Fin