Online Learning Applications
Advertising Project Report

Prof. Nicola Gatti

December 2022

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

POLITECNICO
MILANO 1863

Contents

Chapter 1

Introduction

This project was made for the `Online Learning Applications` class at **Politecnico di Milano** with the supervision of prof. Nicola Gatti during a.a. 2021-2022; the project features a common part to all the project proposals, which is related to *social influencing*, and a proposal-specific argument that, in our case, is *advertising*.

1.1 Overview

The project should model an e-commerce website capable of offering to its customers 5 different possible products labeled as P_i having $i \in (1, 2, 3, 4, 5)$.

Products are displayed inside web pages; a web page may have a *primary product* (which can be bought directly) and two *secondary products* organized as recommendations where the first *secondary product* has higher priority than the second one, moreover, the price for *secondary products* isn't shown but they have the capability of bringing the customer on their respective primary web page upon being clicked.

Every day, customers can land on the primary pages of the various products or on the web page of a competitor.

The website is in need of advertising its own products and has a predetermined *budget* to spend in order to boost the probability of possible customers landing on a page owned by the website.

The focus of this project is to optimize the budget allocated for the **advertisement campaigns** of the 5 products in order to maximize the profit of the website.

1.2 Hypoteses

This section holds all of the general hypoteses that were given or were independently formulated in order to create a feasible and focused project.

This does not include specific hypoteses and assumptions that are strictly related to the computational or numerical side of the project, as they will be discussed in their respective chapters.

- The customer effectively buys products only at the end of their visit to the website.
- A user buys an arbitrary amount of products of a certain type if the price of a single unit of product is strictly under their *reservation price*.

- 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 website offers an infinite number of units for each product and a can customer buy an arbitrary number of them.
- A single user won't open two times the same web page containing the same product during a visit to the website.
- All of the actions performed by the users are perfectly observable by the e-commerce website.
- 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 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.
- Every *primary product* has a **fixed** pair of *secondary products* that will be displayed in the modalities discussed above.
- The price of every product is **fixed** and equal to the margin.
- There is an *advertising campaign* for each product (5 in total), and clicking on an ad brings the user to the web page containing the advertised *primary product*.
- *Bidding optimization* is outside the scope of the project and is not performed.

1.3 Approach

1.3.1 Learners

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.

```
class Learner(ABC):

    """Generic Learner interface for interactive agents capable of learning the budget
    distribution for a set of subcampaigns from the environment they exist in and the
    online feedback returned.
    """

    @abstractmethod
    def learn(
        self, interactions: List[Interaction], reward: float, prediction: np.ndarray
    ):

        """Updates the learner's properties according to the reward received.
        Arguments:
            interactions: the interactions of the users which led to the given
            reward
            reward: the reward obtained from the environment based on the
```

```
            prediction given, needed for the tuning of internal properties done
            by the learner
            prediction: array containing the previous budget evaluation of the learner
        """

        pass

    @abstractmethod
    def predict(
        self, data: MaskedEnvironmentData
    ) -> Tuple[np.ndarray, Optional[List[List[Feature]]]]:

        """Makes an inference about the values of the budgets for the subcampaigns
        utilizing the information gathered over time and the current state of the
        environment
        Arguments:
            data: up-to-date, complete or incomplete environment information that is
                used by the learner in order to make the inference
        Returns:
            a tuple containing a list of values (corresponding to the budgets inferred
            given the knowledge obtained by the learner until now) and a list of
            features (referring to which particular customers were the budgets aimed
            for, if None, the budgets apply to all the customers)
        """

        pass

    @abstractmethod
    def show_progress(self, fig: plt.Figure):

        """Creates a figure and plots showing the status of learning progress
        Arguments:
            fig: a matplotlib figure which will be filled with subplots/plots
        """
        pass
```

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.

1.3.2 Regret

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 while, 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.

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.

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.

Chapter 2

Environment modeling

2.1 Overview

The environment is coded as a constant base upon which we build the entire project. This choice has been derived from two main reasons. *The first one* is that the environment's nature doesn't change in each different step, thus it behaves always in the same way. *The second one* is to better compare the results of different situations without introducing a bias in the inputs we feed the algorithm.

In order to better simulate a real case scenario and to maintain consistency between the different learners, the environment contains all the required functions and data needed by the simulator to compute the outcome of every day given the initial parameters. Each day, the simulator asks for a *budget assigment* and returns the obtained rewards to the learner, which, after adjusting its parameters to maximize the reward for the next predictions, will be ready for the next day.

What will change between steps is how much data the learner will be given as input by the simulator; as a matter of fact, from the learners' perspective, the environment is *"masked"* and it will contain only the known data for each step (different for each point of the project).

2.2 Hypotheses

In this section the specific assumptions related to the environment modelling are listed.

- Each consumer is characterized by 2 binary features, for instance: occupation (student/-worker) and gender (female/male). These features will define the 3 different user classes that we want to target in our advertisement campaign.
- The probability for each class to be able to enter the website is fixed and unknown from the learner's perspective. It can be seen as a percentage over the population that we are focusing with our ads.
- Each user class is distinguishable by an α_i function expressing the ratio of users landing on the web-page where product P_i is shown as the primary one. More clearly, given a campaign every class is characterized by a different profile of α functions.
- The competitor's budget is considered to be constant assuming a *non-strategic player*.

2.3 Model Choice

We chose to model the α functions as *exponential* functions identified by an *reservation price*, an *upper bound* and a *maximum useful budget*. In particular their *upper bound* represents the maximum expected number of interactions possible and the *maximum useful budget* is the amount of budget after which any budget increase would not lead to a ratio increase, meanwhile the *reservation price* is defined as the maximum price at which the user will buy a given product.

Moreover, having three different parameters to tune gives us the possibility to better differentiate each user class.

2.4 Code Analysis

2.4.1 Environment

The environment is composed by different objects and functions which model the users' interactions on the web-page, how users react given different budgets and how the final reward is obtained from a day of interactions.

1. Environment Data

   These two data classes contain all the relevant information for the environment. Most of the parameters for the Environmnet Data class are completed by default at creation time, however, each one of them can be specified when the class is constructed to create a custom environment. Notably, the Environmnet Data class can be passed to the function *get_day_of_interactions* which generates a whole day of interactions from the specified environment.

```python
@dataclass
class EnvironmentData:

    """Dataclass containing environment values. Should be constructed passing
    the rng parameter, which is the only one not defined by default. The other
    parameters are set up correctly by the dataclass constructor, with default
    values as shown below. Setups correctly everything about the environment.
    After constructing this class, it can be passed to the function
    get_day_of_interactions. To construct it with different values they can
    simply be specified when the class is constructed.
    """

    # 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]]

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

2. Masked environment

   In addition, the Masked Environment Data class is a parallel data class w.r.t. Environment Data 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.

3. Modelling user interactions

   With the aid of the functions *simulate_interaction* and *generate_interactions* we can compute a single set of interactions starting from a page, returning a set of intermediate results called `blueprints` which can be used to generate actual interactions following the user behavior defined in section Overview.

4. Overall result of a day

   The *get_day_of_interactions* function is the able to generate a whole day of interactions utilizing the aforementioned functions and accepting a budget parameter obtained by the learner through the simulation and used to influence the distribution of users on different pages; it also needs the *Environment Data* and the total population of visitors of the e-commerce website for that day.

It is worth to note that the Environment isn't necessarily deterministic, in fact, 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.

2.4.2 Simulation

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. It mainly uses functions already definded in the environment to compute the rewards and to adjust the parameters as the learner improves.

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.

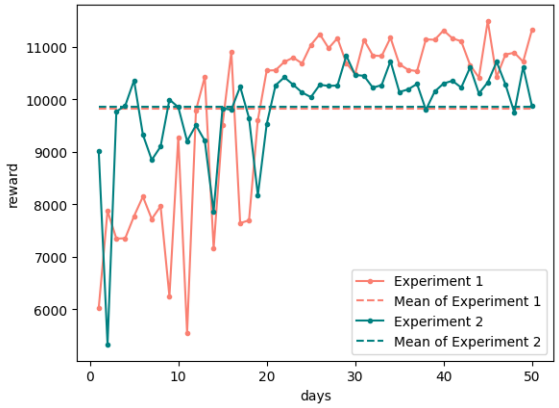
The main method of interaction with a simulation is the `simulate` function, which runs the simulation for a specified number of days, this preserves the status of the simulation, enabling consecutive calls of this function in order to simulate multiple batches of days step-by-step. Inside the simulation module exist some quality-of-life functions such as `create_n` and `simulate_n`, which respectively create and call simulate on batches of n simulations; this is useful to collect average results over a large number of runs.

```python
def create_n(
    rng: Generator,
    env: EnvironmentData,
    step: Step = Step.ZERO,
    n: int = 2,
    n_budget_steps: int = 20,
    population_mean: int = 100,
    population_variance: int = 10,
    **learner_params,
):

    """Helper function that simplifies the creation of multiple equal simulations
        at once
    Arguments:
        n: number of simulations to create
        rng: randomness generator
        env: environment where the simulation is going to be run
        step: step number of the simulation, related to the various steps requested
        by the project specification and corresponding to which properties
        of the environment are masked to the learner and to which learner is going
        to be instantiatiated
        n_budget_steps: number of steps in which the budget must be divided
        population_mean: expected value of the number of new potential
        customers every day
        population_variance: variance of the daily number of potential customers
        learner_params: various parameters used to created the selected learner
    Returns:
        A list of new simulations with the parameters specified
    """

    return [
        Simulation(
            rng,
            env,
            step=step,
            n_budget_steps=n_budget_steps,
            population_mean=population_mean,
            population_variance=population_variance,
            **learner_params,
        )
        for i in range(n)
    ]

def simulate_n(
    simulations: List[Simulation],
    n_days: int = 100,
    show_progress_graphs: bool = False,
):

```

```
    """Helper function that simplifies the act of running multiple simulations with
        the purpose of visualization of the results
    Arguments:
        simulations: list of simulation objects that will be run
        n_days: number of days to run each simulation for
        show_progress_graphs: if set to True, will visualize the learner progress graphs
        (if implemented) at each iteration
    Returns:
        A list containing all the collected rewards obtained by running all
                the simulations.
    """

    rewards = []

    for sim in simulations:
        sim.simulate(n_days, show_progress_graphs)
        rewards.append(sim.rewards)

    return rewards
```

2.4.3 Simulation example

Example of two simulation runs for learners in an environment with total_budget = 400 and population_mean = 1000.



Chapter 3

Optimization algorithm

3.1 Problem Formulation

Our problem, since the bidding part is out of the focus of the project, can be reduced to finding the optimal budget for each subcampaign in order to maximize the profit.

The profit is defined as the difference between the expected margin and the budget spent in advertising: in practice we can calculate it by multiplying the quantity that has been bought for the price of the respective product since in the project's assumption the price is equal to the margin.

The number of clicks is represented by α_i which are influenced by how much of the budget we invest on that particular subcampaign. Basically, it is a maximisation 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:

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

Where:

- F represents the profit that needs to be maximized with respect to the daily budget
- α_i are the value per click, set equal to the expected margin from landing to the corresponding product.
- B is the total budget that can be used.
- x_i are the amount of money spent on the sub campaign ads for each product P_i.

We can use a dynamic programming approach to optimize this function considering that all the parameters are known, in this way we can find the best way to spend the budget, for every feasible value of the budget.

Furthermore we can observe that it's possible to neglect the direct dependence of the margin from the budget cost since it wouldn't affect the result of our optimization approach.

3.2 Code Analysis

We solve the budget assignment problem over the value matrix representing our optimization problem by using the *budget_assignment* function. The matrix is assumed to be divided into N rows and M columns. In the rows we have the sub campaigns, in the columns we have the values of the daily budget described by fractions from 0 to B, where B is the max budget. The function considers that the sum of the returned allocations cannot exceed the previous column. The value in each cell is the reward of assigning budget j to campaign i and return a vector where row i has the index of selected column j of C.

```
def budget_assignment(c):
    """Solve the budget assignment problem over the value matrix c.

    The matrix c is assumed to be divided into n rows (campaigns) and m columns
    (budget fractions). As the columns describe budget fractions from 0 to B,
    where b is the max budget, the function only needs to consider that the sum
    of the returned allocations/selections cannot exceed m - 1. To put it
    another way, column j reprereesents the value of allocating j * (b / m) of
    the budget to the campaigns.

    Arguments:
        c: numpy matrix (nxm) where the value is the reward of assigning budget
           j to campaign i

    Returns:
        a numpy vector where row i has the index of selected column j of C
    """

    n, m = np.shape(c)
    if m == 0 or n == 0:
        return np.array([])

    # Setup a dp table (which will contain the computed cumulative value of a
    # certain allocation) and a table to store the allocations themselves. We
    # initialize both with the first input, as when there is nothing before us
    # we will always allocate the most possible to the single campaign.
    dp = [c[0]]
    allocs = [np.arange(m)]

    for i in range(1, n):
        next_dp_row = []
        next_alloc_row = []
        for j in range(m):
            # We reverse the previous row of the dynamic programming storage, to
            # easily compare the options we have by summing across the vectors
            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)
```

```
54
55      # We get the best allocation from the last row of the dp table
56      best_dp_index = np.argmax(dp[-1])
57
58      # As we stored the allocation of what the dp would mean, we now need to
59      # trace back through the allocs array to create the final allocation
60      last_alloc = allocs[-1][best_dp_index]
61
62      # A list of the final allocations that will give the optimal budget
63      # utilization
64      final_allocs = [last_alloc]
65      remaining_budget = m - last_alloc
66
67      # This will loop backwards through allocs, excluding the last row, so from n - 2 to 0
68      for i in range(n - 2, -1, -1):
69          # Based on the remaining_budget, we take the max of the remaining
70          # possible cumulative values that we have in the dynamic programming
71          # table
72          sub_dp = dp[i, :remaining_budget]
73          sub_best_index = np.argmax(sub_dp) if len(sub_dp) > 0 else 0
74
75          # Convert the index we chose into the actual allocation that was done
76          # for the given sub_dp index
77          next_alloc = allocs[i][sub_best_index]
78          final_allocs.append(next_alloc)
79
80          # As we might have taken more off the budget, we need to adjust it
81          remaining_budget = remaining_budget - next_alloc
82
83      # As we constructed the the final allocation from the back to front, we need
84      # to reverse it before returning it
85      return np.array(list(reversed(final_allocs)))
```

3.3 Solving the Budget Assignment Algorithm

As a part of the optimization problem we utilize a dynamic programming algorithm to solve a discretized budget assignment problem. Assigning the budget in the most profitable way is similar to solving the multidimensional knapsack problem.

By first creating a matrix over the value of assigning a certain budget to a given sub-campaign. The rows of the matrix correspond to the different sub-campaigns and the columns corresponding to the different budget levels. For the algorithm to work, it is important that the columns are separated by a constant step size. As a simplification we have in our case then determined that column j signifies a budget of $j * (B/M)$, where B is the total budget and M is the amount of columns. Put simply, the columns represent a budget fraction of the total budget.

Given this matrix, the algorithm will then find the best allocation of the budget to maximize the total value. By iterating over the rows of the matrix, and in essence taking more and more sub-campaigns into consideration, two new tables are created which store both the highest value so far and the needed allocation to achieve that value. E.g. when calculating the value for budget fraction $\frac{1}{2}$ when only the first sub-campaign is considered, the full amount of the current fraction would always be assigned to current, and only, campaign. However when the second campaign is introduced, all possible combinations of assignments between the two are considered. In that case there might be that allocating 0 to the first and $\frac{1}{2}$ to the second gives the highest value. Then the value of this allocation is stored in the dynamic table at the second row and column of the budget fraction $\frac{1}{2}$. Lastly the second table is populated with the index of the allocation made, so in this case the index of the chosen budget fraction.

After constructing the two dynamic tables above, the last rows then consist of the situation in which all sub-campaigns are considered, hence by choosing the biggest value in this row, the best budget assignment for the entire problem is found. Using the table with the allocation indices the amount of the budget for the given campaign is stored, and the remaining budget is updated. I.e. if allocated half of the budget for the last campaign, we want to best distribute the remaining half of the budget to the other sub-campaigns.

Chapter 4

Optimization with uncertain α functions

4.1 Contextual hypotesis

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.

4.2 Algorithm

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.

4.2.1 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:

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

The regret bound for algorithms that utilize gaussian processes in an advertising environment is identified as follows:

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

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

4.2.2 Algorithms outline

The main two MAB algorithms that we are going to use present some minor variations when used in conjunction with Gaussian Processes; we can consider the GPTS algorithm as a baseline for the GPUCB1 since it directly uses the GPs' features to gather samples, while in the GPUCB1 we need to build a confidence bound before.

GPTS is a variant of TS implemented using Gaussian Processes that is implemented as follows:

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

$$\tilde{\theta_a} \leftarrow \text{ Sample } (\mathbb{P}(\mu_a = \theta_a))$$

- 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.

Code snippets for updating the model and returning an estimation:

```
def _update_model(self):
        x = np.atleast_2d(self.pulled_arms).T
        y = np.array(self.collected_rewards) / self.normalize_factor
        self.gp.fit(x, y)
        self.means, self.sigmas = self.gp.predict(
                np.atleast_2d(self.arms).T, return_std=True
        )
        self.sigmas = np.maximum(self.sigmas, 1e-2)
```

```
def estimation(self):
        return self.rng.normal(
                self.means * self.normalize_factor, self.sigmas * self.normalize_factor
        )
```

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. The arm choice is implemented as:

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

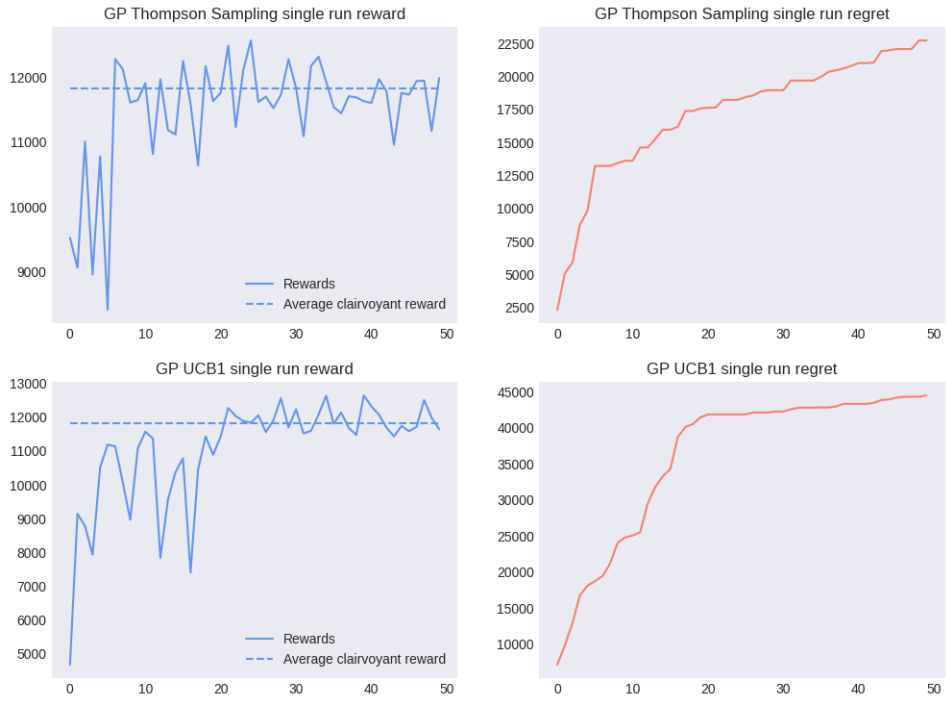
Code snippet that calculates the upper bound:

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

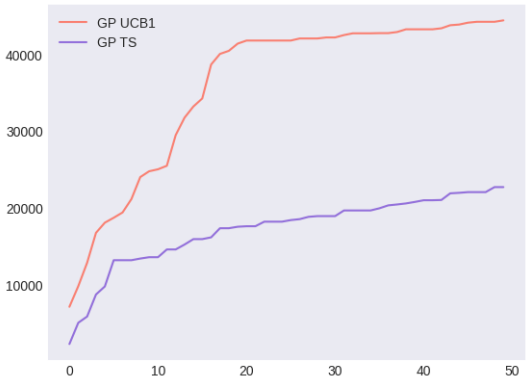
4.3 Results

4.3.1 Single run reward and regret

Thompson Sampling and UCB

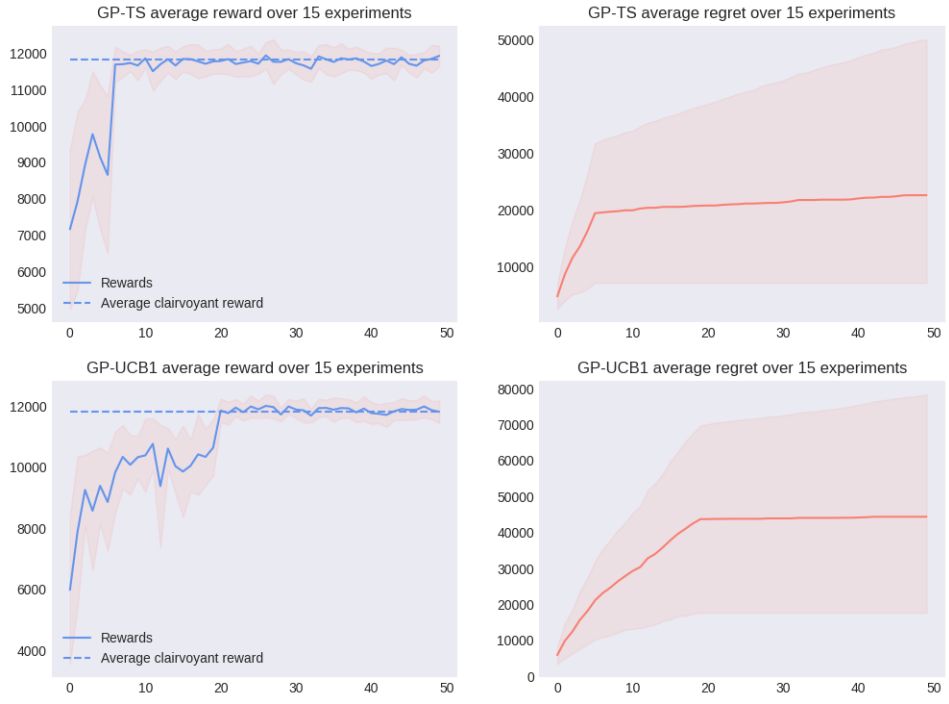


Regret comparison

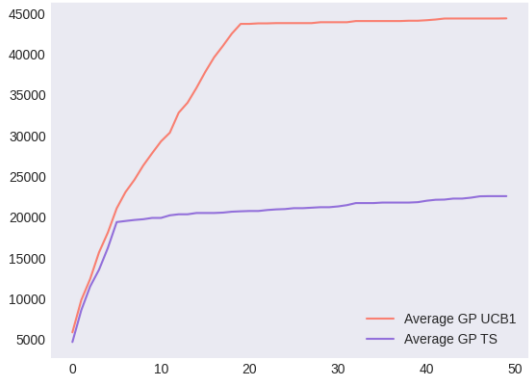


4.3.2 Average regret and reward

Thompson Sampling and UCB



Average regret comparison



4.3.3 Conclusions

Overall we can observe more instability but a faster convergence in the TS algorithm w.r.t to the UCB approach. However, both algorithms are able to 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

Chapter 5

Optimization with uncertain number of sold items

5.1 Contextual hypotesis

In this case the e-commerce website doesn't register neither the units sold for each product nor the class parameters for the users. Since this isn't a change that affects the enviroment directly, there will not be a separate mask to hide the *units sold* to the learner, therefore the extra information will just 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.

5.2 Algorithm

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.

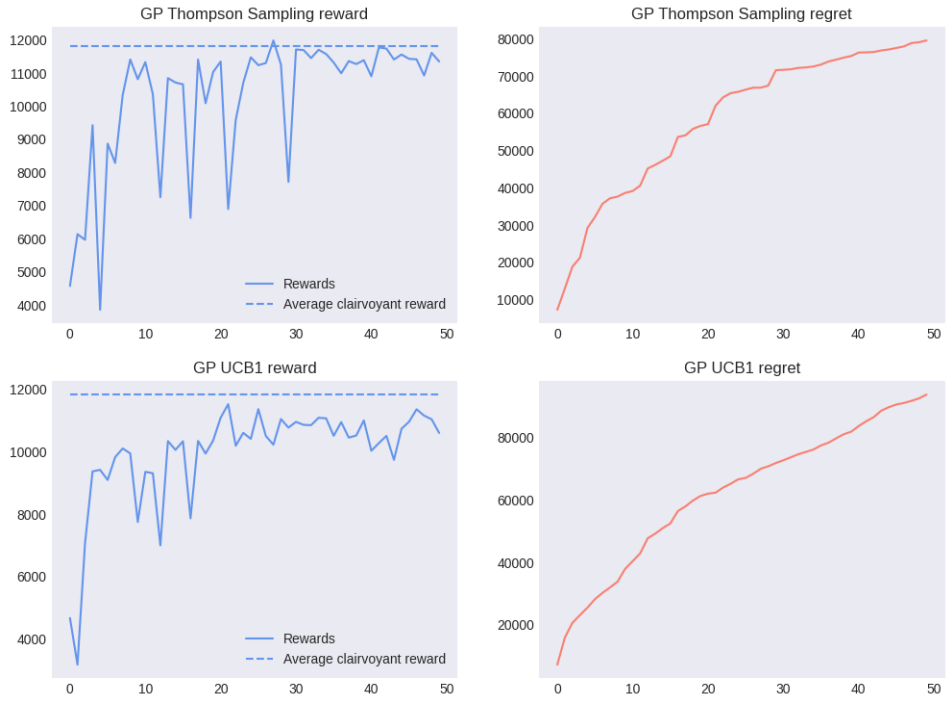
However, in this case, the `learn` function is only allowed to gather information from the generic reward obtained for the day.

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

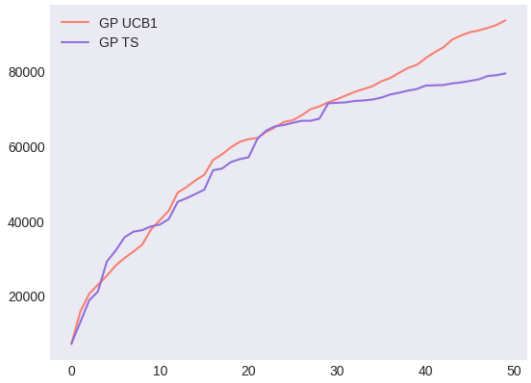
5.3 Results

5.3.1 Single run reward and regret

Thompson Sampling and UCB



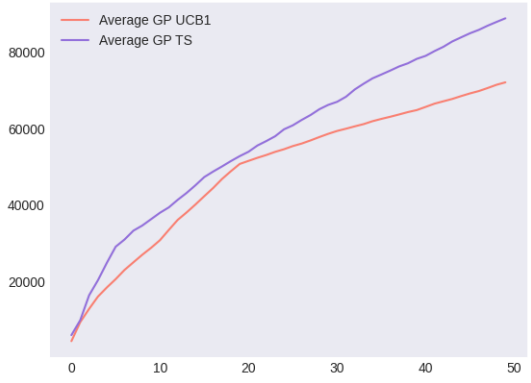
Regret comparison



5.3.2 Average regret and reward

Thompson Sampling and UCB

Average regret comparison



5.3.3 Conclusions

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

Both the TS and UCB approaches are not guaranteed to find the optimal arm as sometimes they seem to settle on a suboptimal solution, however we can see that the TS performance is much more unstable overall, this is also reflected in the average regret.

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.

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

Chapter 6

Optimization with uncertain graph weights

6.1 Problem

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.

6.2 Implementation

6.2.1 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`. The rationale for our graph influencing approach was based on determining how much each product was valuable to invest budget in by following activation patterns on the estimation of our graph, which contained our approximate weights.

In the end we opted for a more experimental approach that better exploited all of the information at our disposal in this particular step: 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 following the behavior observed on real interactions.

6.2.2 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.

6.2.3 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.

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.

6.2.4 Relevant code

Implementation of the `predict` function:

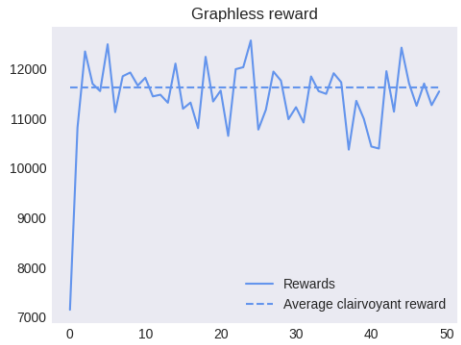
```
def predict(
        self, data: MaskedEnvironmentData
) -> Tuple[np.ndarray, Optional[List[List[Feature]]]]:
        budget_steps = np.linspace(0, data.total_budget, self.n_budget_steps)

        # Sample current estimation of graph
        graph = self.graph_estimation()

        best_allocation = find_optimal_superarm(data, budget_steps, custom_graph=graph)

        return budget_steps[best_allocation], None
```

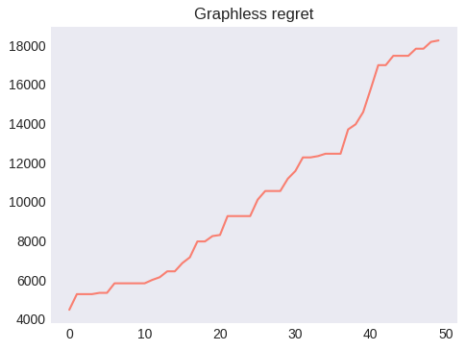
Implementation of the `learn` function:

```
def learn(
        self, interactions: List[Interaction], reward: float, prediction: np.ndarray
):
        for interaction in interactions:
                remaining_edges = [
                        (product, next_)
                        for product, nexts in enumerate(self.next_products)
                        for next_ in nexts
                        # Only add edge to remaining if we visited the page, otherwise
                        # we got no information about those edges
                        if interaction.items_bought[product] > 0
                ]
                for edge in interaction.edges:
                        s, t = edge
                        self.alpha_param[s, t] += 1
                        remaining_edges.remove(edge)

                for edge in remaining_edges:
                        s, t = edge
                        self.beta_param[s, t] += 1
```

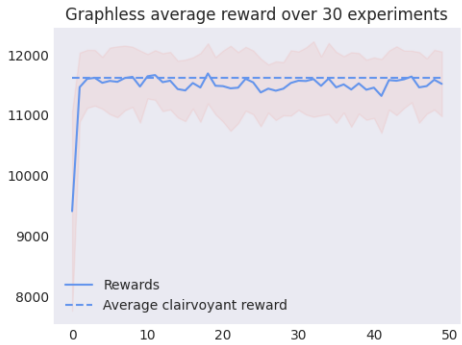
6.3 Results

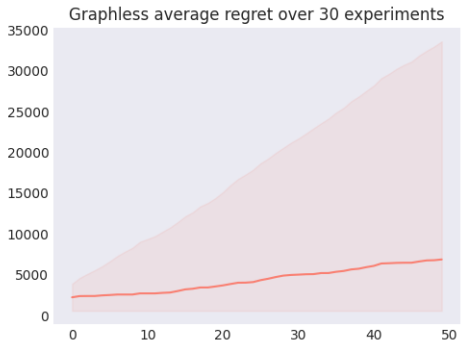
6.3.1 Single run reward and regret





6.3.2 Average reward and regret





6.3.3 Conclusions

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 has access to 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

Chapter 7

Optimization with non-stationary demand curve

7.1 Environment behavior

7.1.1 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.

7.1.2 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.

In the early stages of the project we noticed that the simulation that we created wasn't really fit to dinamically include abrupt changes in the environment, therefore great effort was spent in completely reworking the interface for the simulation from the ground-up. Besides collateral improvements in reproducibility, results collection and user-friendliness, the new simulation allowed for an *incremental simulation unfolding*, this meant that we were now able to simulate n days, change the environment and simulate another n days. This particular feature revealed itself as fundamental for the upcoming challenges.

7.2 Algorithms

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.

7.2.1 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:

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

7.2.2 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.

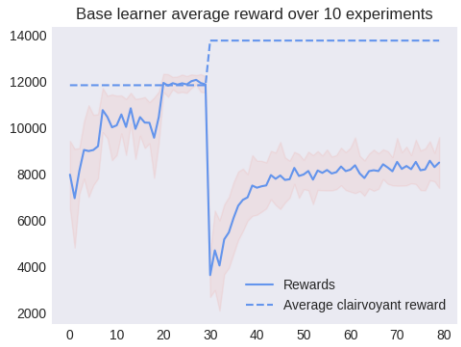
Formally:

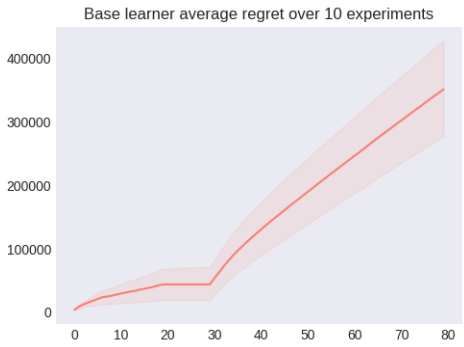
$$\text{RESET learner if } t \geq 2k \ \wedge \sum_{i=t-2k}^{t-k} r_i - \sum_{i=t-k+1}^{t} r_i > \omega$$

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

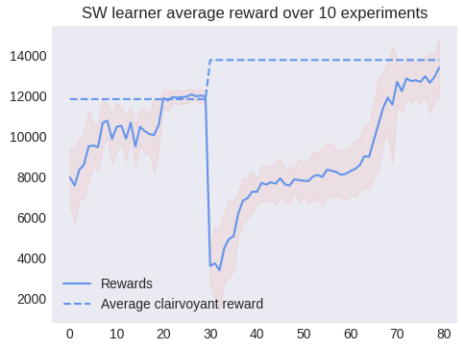
7.3 Results

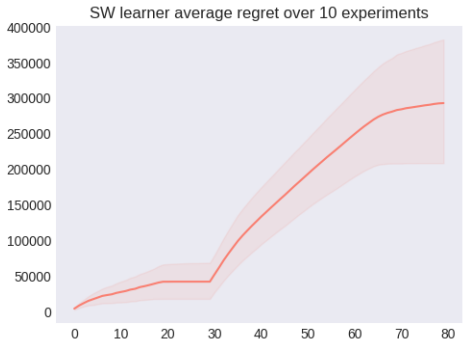
Average reward and regret for the base learner



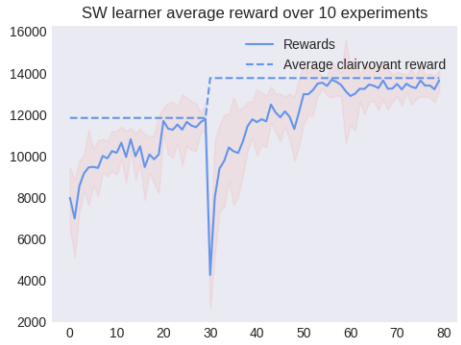


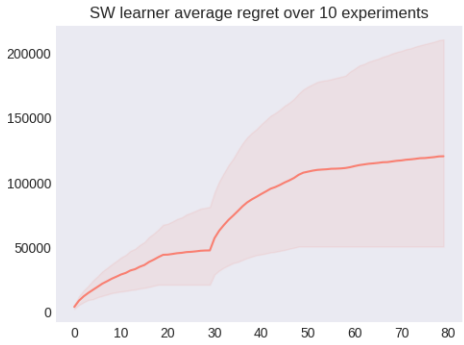
Average reward and regret for the sliding window learner



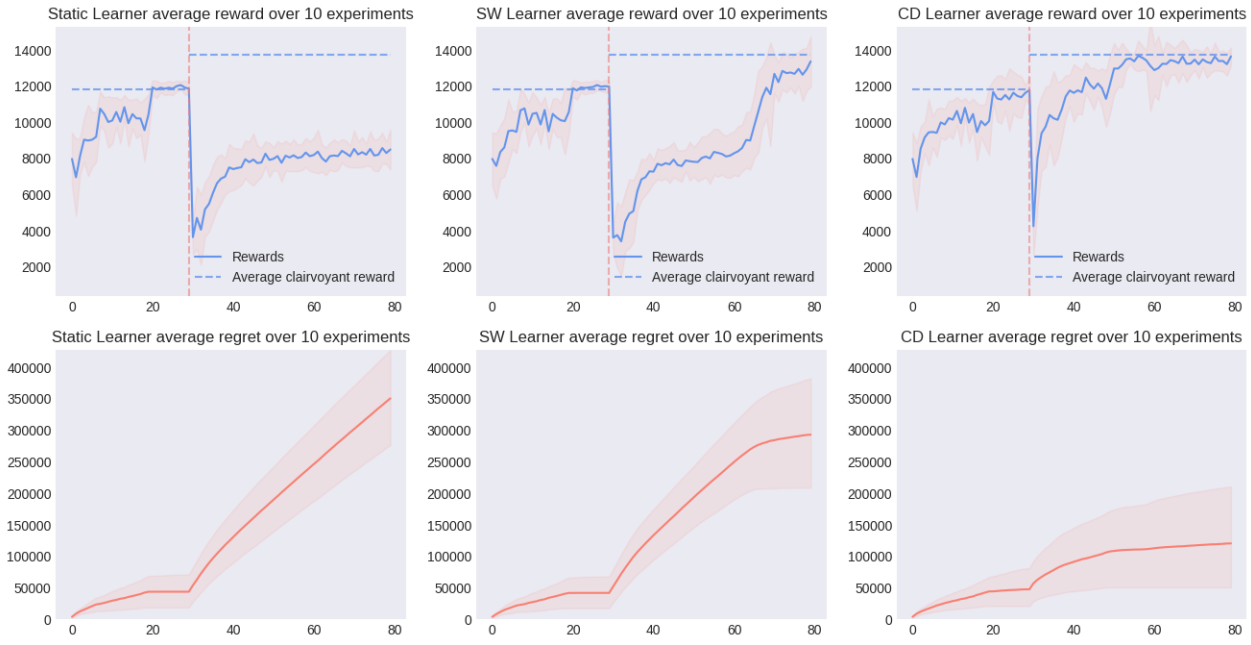


Average reward and regret for the change detection learner





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*; while only UCB algorithms were evaluated and only one breakpoint was placed after 30 days.

7.3.1 Conclusions

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 $O\left(|A|T^{\frac{1+\alpha}{2}}\right)$.

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

Chapter 8

Context generation

8.1 General Remarks

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.

8.2 Assumptions

8.2.1 Implementation challenges

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. 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.

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.

8.3 Implementation

8.3.1 Greedy algorithm

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.
- 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*.

8.3.2 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.

8.3.3 Context utilization

Once a certain number of contexts has been established, a Contextual Learner will be able to assign an Alphaless 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.

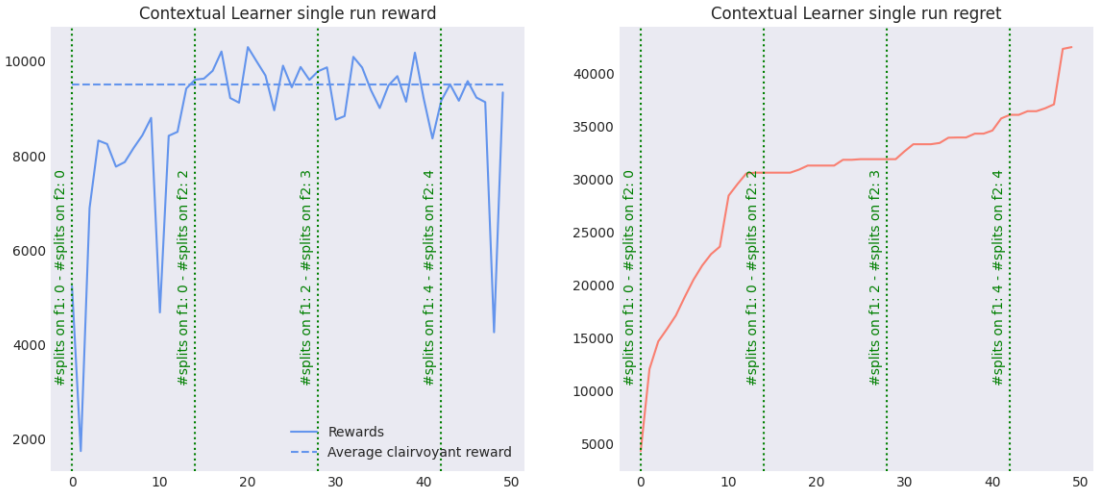
The resulting *superarm* is then passed to the simulation which, in return, generates the interactions for the day by weighting the different classes according to the disaggregated predictions.

```
1  def predict(self, _) -> Tuple[np.ndarray, Optional[List[List[Feature]]]]:
2          aggregated_budget_value_matrix = [
3                  np.array(learner.predict_raw(self.env)) for learner in self.learners
4          ]
5          aggregated_budget_value_matrix = np.vstack(aggregated_budget_value_matrix)
6          best_allocation_index = budget_assignment(aggregated_budget_value_matrix)
7          best_allocation = self.budget_steps[best_allocation_index]
8
9          budgets = []
10         features = []
11         for i, learner in enumerate(self.learners):
12                 lower = i * self.n_products
```

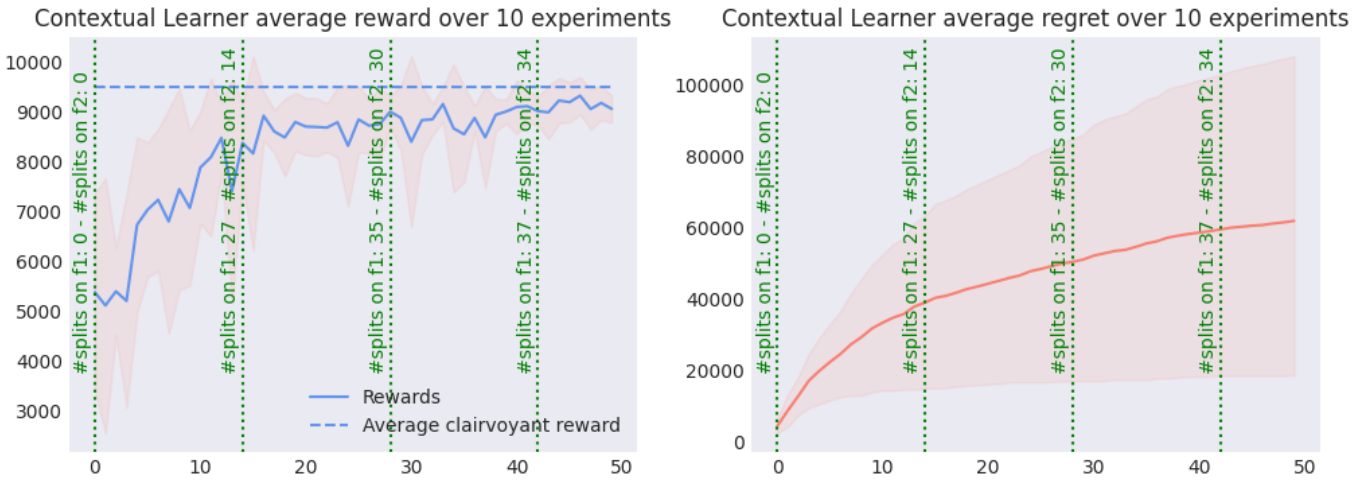
```
                upper = (i + 1) * self.n_products
                budgets.append(best_allocation[lower:upper])
                if self.contexts[i].features:
                        features.append(self.contexts[i].features)
        return budgets, features
```

8.4 Results

Reward and regret for the contextual learner



Average reward and regret for the contextual learner



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

8.4.1 Conclusions

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*. Furthermore, by looking at the `example_environment` structure, it comes clear why the contextual learners seem to prefer splitting on the first binary feature: it creates an evident division between the first class and the other two classes while splitting on the second feature first would create hybrid contexts.

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

	Reward	Regret	Deviation
	μ	μ	σ
Contextual learner	9065.30	61719.2	273.67

Chapter 9

Conclusions

Even though some of the proposed problems weren't fully solved, most of the results that we found were in with the theoretical guarantees of the literature. The specifications were at times confusing and misleading, this was a deliberate design choice as it would be closer to a "real life scenario", however this caused several difficulties in the understanding process of the project and was ultimately one of the concurring causes for the deadline delays.

Overall, great effort and time was spent on this project as it revealed itself as a thougher challenge than what we had anticipated, we are mildly satisified with our work as we collectively think that there is still vast room for improvement in several aspects of the project.