POLITECNICO

MILANO 1863

ONLINE LEARNING APPLICATION
ACADEMIC YEAR 2021 - 2022

Pricing & Social Influence

Sofia MARTELLOZZO   Vlad Marian CIMPEANU   Lorenzo ROSSI
(10623060)   (10606922)   (10698834)

Professor

Nicola GATTI

Contents

Introduction

Nowadays one big problem of e-commerces is to allocate the best price to its products so that, the seller can maximize its revenue.

The main issue is that increasing the price of a product leads to fewer people interested in that product, thus increasing the price is not necessarily beneficial to the seller. In contrast, decreasing the price will increase the number of people interested in the product, but the revenue will be of course sub-optimal.

In order to maximize the revenue, we can analyze the demand curve of a given product, which is a graphical representation of the relationship between the price p_i of a good or service i and the quantity demanded $q_i(p_i)$ for a given period of time, and find the price $\hat{p}$ such that:

$$\hat{p} = \arg\max_p (pq(p))$$

Unfortunately, in real-world problems, the demand curve is not available, furthermore, we need to estimate this curve by interacting with the environment. One main problem of interacting with an unknown environment is that exploration costs a lot of money, so we want to find the best prices in the shortest amount of time to decrease the regret.

In order to do so, we can use reinforcement learning techniques such as Multi Armed Bandit (MAB) algorithms.

Practical example

In this project, we want to study the case of a new e-commerce entering the market called ANS[2] that sells skateboarding clothes. More precisely, it is going to sell unisex t-shirts, hoodies, t-shirts, shoes, and shirts.

For simplicity's sake we can assume the website can sell an unlimited number of units without any storage cost whose goal is to minimize the cumulative regret while learning.

The website of the vendor is structured as follows: in every webpage, a single product, called primary, is displayed together with its price. The user can add a number of units of this product to the cart. After the product has been added to the cart, two products, called secondary, are recommended. When displaying the secondary products, the price is hidden. Furthermore, the products are recommended in two slots, one above the other, thus providing more importance to the product displayed in the slot above. The website will propose only products that the customer has never seen before. If the user clicks on a secondary product, a new tab on the browser is opened and, in the loaded webpage, the clicked product is displayed as primary together with its price.

The choice of the products that must be recommended to the user is already fixed by the business unit as presented in Figure 1.

One main consideration we want to make is that the customer may buy different products during a visit, thus the price for a specific product may influence the total income generated by the customer. For instance, let us assume that a customer lands on the webpage displaying a t-shirt: if the price is too high, the probability to buy that product is lower, but not only, also the probability to see the secondary products is lower, so it will decrease the probability that a customer visits and buys new products. In conclusion, when we choose the price for a specific product we have also to consider the indirect reward it will generate.

Figure 1: Recommendation graph set by the business unit.

1 Step: Environment

The main aim of the environment is to simulate a real-world scenario. To simulate all the components we divide the model into various classes.

- Environment class: It is the wrapper that manages the environment and all its functions. There are two specializations of this class created for some specific use cases (which are EnvironmentContextual and EnvironmentNonStationary).

- Simulator class: It manages the simulation of the customers' interactions.

- Customer class: It contains all the information that defines a type of customer.

1.1 Parameters

The environment has a lot of parameters and each of them has a direct and significant impact on the behavior of the model. Some of the parameters are specifics of the environment:

- `customers_distribution` is a list of 4 floating values, that sum to 1. It indicates the probability of each type of customer appearing.

- `customer_per_day` is the average number of customers in a day.

- `variance_customers` is the standard deviation of the number of customers in a day.

- `products_graph` is the graph that indicates which is the primary and secondary product.

- `p_lambda` is the probability of observing each slot. The first value is 1, while the second is a number smaller than 1.

- `prices` indicates the reward for each product and each price level. Therefore, it is a 4 x 5 matrix.

Additionally, some parameters are specifics of the customer:

- `features` is the pair of binary features. In our case, the first feature is associated with gender whereas the second one with the age (for simplicity young/old).

- `alpha` is a vector that contains the probability distribution of starting from each product.

- `buy_distribution` is a matrix containing for each pair (product, price) that defines the probability of buying products.

- `num_prods_distribution` is a matrix containing for each pair (product, price), and it controls the number of units the customer is likely to buy for that specific product. In this case, we have decided to model the customer's behavior with a geometric distribution. Thus, they are the p parameters of the geometric distributions, which is the inverse of the means. The main idea behind the geometric distribution is that is a discrete distribution and is monotonically decreasing: the higher the number of products bought, the lower will be the probability to buy that amount of units.

- `click_graph` is the probability of clicking the product given a selected price and type of product.

1.2 Learner interaction

The main aim of the learner is to minimize the cumulative regret by selecting the best price levels. Therefore, at the beginning of each day, the learner selects the price levels (which is the super arm), and then at the end of the day, it will obtain a report containing all the information about the customer activities, i.e. number of times bought, number of products seen, number of customers.

1.3 Customer interaction

The customer interaction works in the following manner:

1. Depending on the `alpha` distribution a starting point is randomly chosen.

2. The customer opens the page and she will buy one or more products with a probability depending on `buy_distribution`. If she does not buy the simulation stops, otherwise the number of items bought is sampled from a geometrical distribution.

3. Then, with a probability that depends on λ and the `click_graph`, she explores a different product. However, if she has already seen this product, she will not open that page.

From step 3 to step 6 the main assumption is that we have different classes of customers interacting with the website, and each of them has a different behavior. For each step, we know the distribution of the customers and some characteristics of them (for instance the mean of the number of items a customer buys for a specific product), but the website is not able to identify the customer is interacting with it, thus we are not able to estimate the unknown parameters for each customer but only an aggregated estimate.

1.4 Selection of the super arm

Since we have to take into consideration also the indirect income generated by other products that are bought after buying the first product, we have to find the combination of prices such that it maximizes the overall income considering the indirect margins, thus we have to solve a combinatorial problem to select the best super arm to play.

In order to do so, we have to compute the believed expected reward $\mathbb{E}[r_a]$ for each superarm a and choose the arm with the highest expected reward. We compute the expected reward as follows:

$$\mathbb{E}[r_a] = \sum_{i \in \mathscr{C}} r_{a,i} w_i$$

where $\mathscr{C}$ is the set of indexes of customers, $r_{a,i}$ is the expected reward given the super arm a for customer i and w_i is the probability to see customer i.

1.4.1 Monte Carlo methods

A way to compute the expected reward for a given customer i is to use a Monte Carlo simulation. Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle.

Basically, we have 5 seeds (the products) and we run N simulations starting from each seed. A seed is a starting node in a graph, whose edges $e_{i,j}$ represent the probability of clicking product j given that product i has been bought, whereas each node has an activation threshold that coincides with the probability of buying that product.

The simulation works as follows:

1. Explore the simulation graph in a depth first search tree fashion. If a node has been already visited, it can not be reached anymore.

2. When selecting a node j, draw a sample x from a Bernoulli distribution $\mathscr{B}e(t_{i,p,j})$ where $t_{i,p,j}$ is the probability that customer i buys product j at price p. If $x < t_{i,p,j}$, activate node j, otherwise stop the branch.
   We can keep track of the number of times a node j has been activated with the variable Λ_j.

3. When expanding an active node i towards node j, draw a sample x from a Bernoulli distribution $\mathscr{B}e(e_{i,j})$. If $x < e_{i,j}$ move towards node j, otherwise stop the branch.

Once a node is activated we draw a sample n from the distribution of the number of items bought for that node (product). Every time a product is bought, we update the number of times that product has been bought:

$$n_{a,p} = n_{a,p} + n$$

where $n_{a,p}$ is the current number of units bought for product p given the super arm a.
Here, the code for a single simulation:

```python
def shopping_dfs(self, primary, displayed_primary, report, super_arm, c):
    displayed_primary[primary] = True
    report.seen(primary)
    if np.random.random() < c.get_probability_buy(primary, super_arm[primary]):
        amount = c.get_num_prods(primary, super_arm[primary]) #1
        report.bought(primary, amount)
        click_prob = [c.get_probability_click(primary, secondary) for secondary in
        ↪   self.products_graph[primary]]
        for secondary, edge_prob, lamb in zip(self.products_graph[primary],
        ↪   click_prob, lamb_SLOTS):
            if not displayed_primary[secondary] and np.random.random() < lamb *
            ↪   edge_prob:
                report.move(primary, secondary)
                self.shopping_dfs(secondary, displayed_primary, report, super_arm, c)
```

Finally, we compute the expected reward as:

$$r_{a,i} = \frac{1}{N} \sum_{p \in \mathscr{P}} \alpha_{i,p} a_p n_{a,p}$$

where $\mathscr{P}$ is the set of indexes of the products, $\alpha_{i,p}$ is the probability for customer i to start from the seed p and a_p is the price selected for the product p. We can also compute the activation probability (probability that a specific item is bought) of each product as:

$$\pi_{p,i} = \frac{1}{N} \sum_{p \in \mathscr{P}} \alpha_{i,p} \Lambda_p$$

Thanks to the following theorem we have some theoretical guarantees on the accuracy of this method.

Theorem With a probability of at least $1 - \delta$, the estimated activation probability of every node is subject to an additive error of $\pm \epsilon n$ when the number of repetitions is:

$$R = O(\frac{1}{\epsilon^2} log(\mid S \mid) log(\frac{1}{\delta}))$$

where S is the number of seeds (5) and n is the number of nodes in the graph (still 5).

In our case, if we want to have an additive error of $\epsilon n = 0.1$ ($\epsilon = 0.02$) with a probability of 90 % ($\delta = 0.1$), we need to run $R = 1748$, thus the number of simulations for each product for each customer is $N = 350$. In conclusion, this method is stochastic and very noisy, and to obtain a decent estimate we need to run a massive number of simulations which makes the simulation process astonishingly slow.

Given the size of the parameters for this problem, we can afford to compute the exact value for the expected value for a given arm in a reasonable time with a different approach.

1.4.2 Dynamic programming approach

To overcome the limitation of the Monte Carlo simulation we develop a dynamic programming solution that returns the expected number of items bought. This method has a time complexity of $\Theta(2^N NM)$, where N is the number of items and M is the number of types of customers, therefore, this solution is only feasible because the number of items is quite small.

```python
def run_dp(self, super_arm):
    ans = 0
    for c, p in zip(self.customers, self.customers_distribution):
        @lru_cache(maxsize=None)
        def dp(primary, mask):
            mask |= 1 << primary
            ans = np.zeros(5)
            ans[primary] = 1 / c.num_prods_distributions[primary][super_arm[primary]]
            click_prob = [c.get_probability_click(primary, secondary) for secondary in
            ↪  self.products_graph[primary]]
            for secondary, edge_prob, lamb in zip(self.products_graph[primary],
            ↪  click_prob, lamb_SLOTS):
                if (mask & (1 << secondary)) == 0:
                    ans += lamb * edge_prob * dp(secondary, mask)
            ans *= c.get_probability_buy(primary, super_arm[primary])
            return ans
        for primary, alpha in enumerate(c.get_distribution_alpha()):
            ans += p * alpha * dp(primary, 0)
    return ans
```

1.5 Assumptions

In this section we list the main assumptions we undertook to face this problem:

- for simplicity we assume that the value λ for the second displayed product is equal to 1 without loss of generality.

- The probability α_0 which is the probability to land on another website is omitted, since it does not add any additional feature to the problem.

- Customers, accordingly to their behaviour can be grouped into three classes:

  1. Boys
  2. Girls
  3. Adults

  Note: this is just an assumption used to model the customer of the environment, the learner is not acknowledged about this assumption.

2 Step: Optimization algorithm

Here we consider the case in which all the parameters are known and the goal is to maximize the cumulative expected reward, following a greedy approach.
The algorithm works as follows:

1. set the prices of all the products with the lowest one

2. collect the reward obtained by increasing, each time, the price of just one product of the original super arm

3. compare the five different configurations obtained with the first one. There could be two cases

   (a) there is an increase of the reward, so select the one that gave the maximum reward (the highest increase) as the best one and repeat the algorithm from point 2

   (b) there is no increase (all the new configurations are worse than the previous one) and stop the algorithm

4. Return the actual best configuration.

For example:
The algorithm starts with the super arm with all the lowest prices for all the products: [00000]
Then explore by pulling the five combinations of superarm, found by increasing just one product at a time:
[00**1**00]
[0000**1**]
[000**1**0]
[**1**0000]
[0**1**000]
Now select the one with the highest reward, and compare also with the original best one. In our case it was: [00001]
Update the new super arm as the best one and start again, looking at the new five configurations available:
[0000**2**]
[00**1**01]
[0**1**001]
[**1**0001]
[000**1**1]
And so on, until no improvement is found.

2.1 Limitations

This type of learner does not directly consider the parameters of a Customer, it just interacts with the environment by selecting the arm to pull at each round and observing the reward given by the environment.
Is guaranteed that the algorithm would not cycle, because it monotonically increases the prices (as well as the cumulative expected margin). On the other hand, there is no guarantee that the algorithm will return the optimal price configuration.
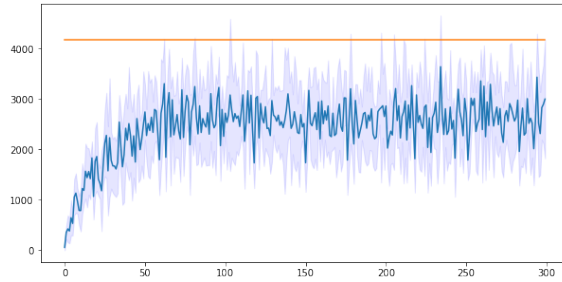
2.2 Results

Figure 2: Reward

As said before here we can see that the algorithm converges to a solution that is not optimal, so return a reward that is lower than the best possible one (clairvoyant)
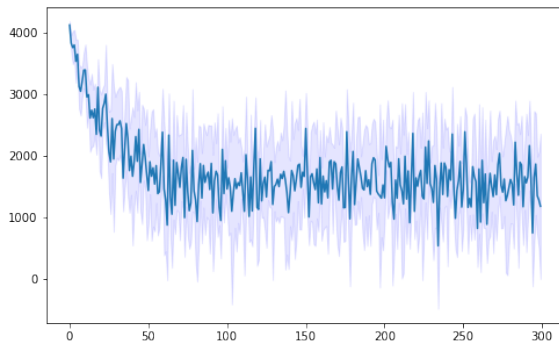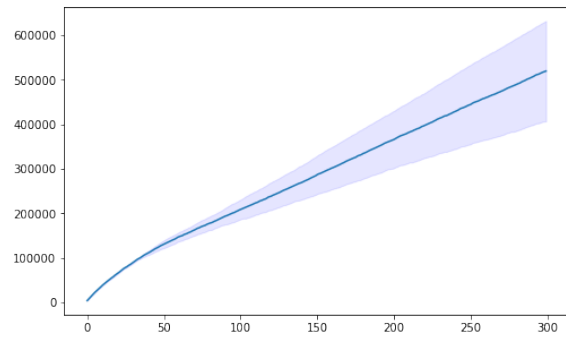

Figure 3: Regret

Figure 4: Cumulative regret

As expected the cumulative reward is linear (Figure 3)

3 Step: Optimization with uncertain conversion rates

The new scenario we have to consider is the one in which only the conversion rates of the customers are unknown. Also, the features of our customers are unknown, so our website is not able to distinguish one from another.

To solve the problem, two bandit algorithms have been developed: UCB-1 (Upper Confidence Bound) and TS (Thompson Sampling). Both select every day the super arm to pull by estimating the conversion rates, with their parameters. After that, they run a Montecarlo simulation for each combination of arms, in order to determine the best super arm (the one that returns the highest reward estimated).

The super arm selected is pulled into the environment to collect the actual reward and the learners' parameters can be updated.

Since the learners are not able to distinguish the different types of customers, the conversion rates estimated each round by the learner are set equal for every type of customer.

The number of daily interactions (customers that visit the e-commerce) is fixed to 100 and the time horizon (the number of days of exploration of the algorithm) to 300. The number of iterations is set to 5 (the number of times the algorithms are executed).

3.1 UCB-1

Upper Confidence Bound is a deterministic algorithm that associates an upper confidence bound to every arm, providing an optimistic estimation of the reward. The steps are the following:

1. Initially the means and the upper confidence bounds of each arm (price) for each product are set to zero and infinite respectively

$$
means = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad upperBound = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}
$$

2. Estimate the conversion rates by the sum of the mean and the upper confidence bound of each arm

```
def estimate_conversion_rates(self):
        return self.means + self.upper_bounds
```

3. Select the super arm that returns the highest reward from the MC simulation and use it to interact with the environment

4. Update the mean parameters of the pulled arm as:

$$
mean[p, a] = \frac{mean[p, a] * seen[p, a] + bought[p]}{tot\_seen[p, a]} \tag{1}
$$

Where:

- mean[p,a] is the actual mean of the product **p** with price **a**

10

- seen[p,a] is the number of times the product **p** with price **a** has been seen until the day before

- bought[p] is the number of times the product **p** has been bought from day zero until now

- tot_seen[p,a] is the number of times the product **p** with price **a** has been seen from day zero until now (so is seen[p,a] plus the number of times this product has been seen today)

5. Update the upper confidence bound parameters of the arm pulled according to the observations of the environment:

$$upper\_bound[p, a] = \sqrt{\frac{2 * \log(tot\_samples)}{seen[p, a]}} \quad (2)$$

Where:

- tot_samples is the total number of times the product **p** has been seen until now

- seen[p,a] is the number of times the product **p** with price **a** has been seen until now

6. Repeat from step 2 until the end of the interactions (300 days)

3.1.1   Results

Even if the algorithm converges to the optimal arm at the beginning of the process, someday it will change the choice of the super arm to pull. This happens because each time an arm is pulled its bound is reduced, whereas at each time step, the upper bound of all arms logarithmically increases, so at some point, the ones less selected will have a higher bound. A bigger bound leads to a higher conversion rate. Due to this reason, the learner will try new super arms that were not pulled for a long time.
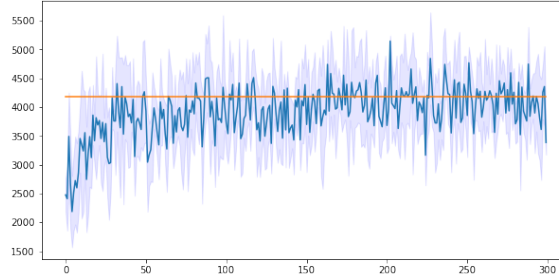
Figure 5: UCB Reward
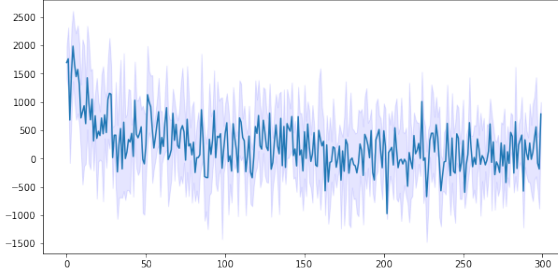
Figure 6: UCB Regret
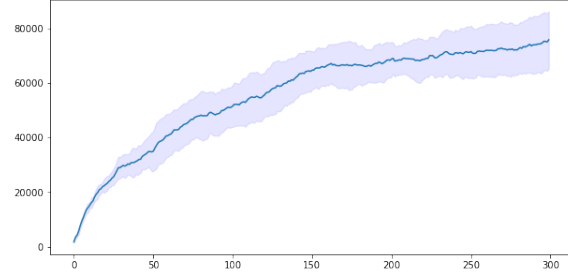
Figure 7: UCB Cumulative regret

3.2 TS

Thompson Sampling is a stochastic algorithm that sets a prior on the expected value for every arm, and selects the arm with the best sample, according to the updated parameters. Those parameters are called Beta parameters and take account of the number of successes (α) and failures (β). The sum of the two parameters is equal to the number of times the arm has been pulled.

Since we are dealing with a combinatorial problem, for each arm, we draw a sample from the conversion rate of the prior distribution and use those samples to evaluate the value of each super arm.

The steps are the following:

1. Set all the α and β parameters equal to one.

$$\left[\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \right]$$

2. Estimate the conversion rate by drawing a sample from a Beta distribution with α and β parameters

```
def estimate_conversion_rates(self):
    return np.random.beta(a=self.beta_parameters[:, :, 0],
    ↪  b=self.beta_parameters[:, :, 1])
```

3. Select the super arm that returns the highest reward from the MC simulation and use it to interact with the environment.

4. Update the α and β parameters according to the observations of the environment:
$\alpha[\text{p, a}] = \alpha[\text{p, a}] + \text{bought[p]}$

$\beta[\text{p, a}] = \beta[\text{p, a}] + \text{seen[p]} - \text{bought[p]}$

- $\alpha[p, a]$ is the alpha parameter for the product **p** with the price **a**
- $\beta[p, a]$ is the beta parameter for the product **p** with the price **a**

- bought[p] is the number of times the product **p** has been bought until now

- seen[p] is the number of times the product **p** has been visualized by the customers

5. Repeat from step 2 until the end of the interaction (300 days)

3.2.1 Results

As we can see below the algorithm converges to the optimal super arm very quickly, so the regret goes to zero in the very first days. This is what we expected when there is a prior information that influences the choice of an arm.
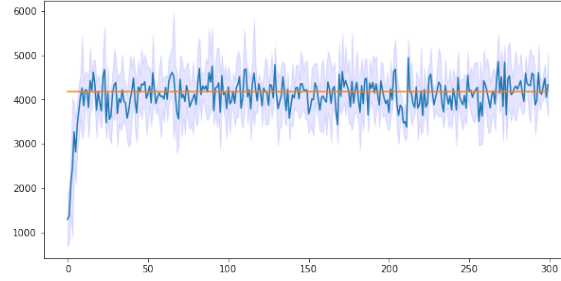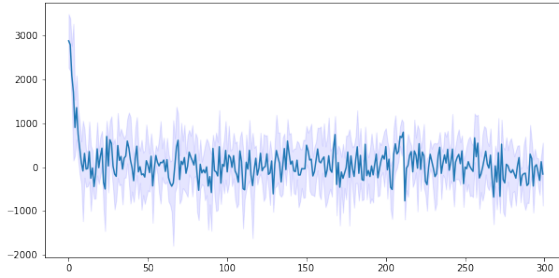

Figure 8: TS Reward

Figure 9: TS Regret
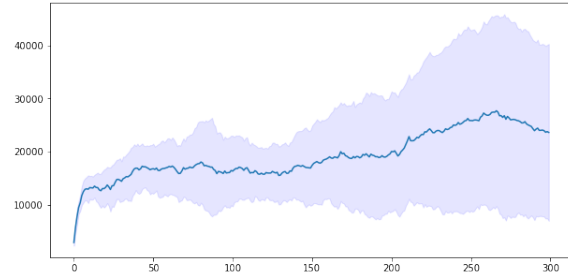
Figure 10: TS Cumulative regret

13

4 Step: Optimization with uncertain conversion rates, α ratios, and number of items sold per product

Here the problem is the same as in the previous step, with the only difference being that also the alpha ratios and the number of products sold for each price are unknown. Since the learners are not able to distinguish the different types of customers, initially the alphas values (probability to start the interaction from a specific product) are uniformly distributed ($[0.2, 0.2, 0.2, 0.2, 0.2]$) and the number of products bought for each price is set all to 1.
The same algorithms as before have been developed, with the additional calculus to estimate the two parameters, now uncertain.

4.1 UCB-1

The algorithm repeats the same operations as in the previous chapter from step 1 to step 5. Then, it estimates the two parameters:

- estimate alpha ratios $\alpha_{i,t}$:

$$\sigma_{i,t} = \sigma_{i,t-1} + starts_{i,t}, \forall i \in \mathscr{P} \tag{3}$$

$$\alpha_{i,t} = \frac{\sigma_{i,t}}{\sum_{j \in \mathscr{P}} \sigma_{j,t}}, \forall i \in \mathscr{P} \tag{4}$$

  Where:

  - $\sigma_{i,t}$ is the number of starts from product i has been observed until time t
  - $starts_{i,t}$ is the number of starting times observed at time t for product i
  - $\mathscr{P}$ is the set of indexes for the products

- estimate number of products sold (version 1):
  with this first approach, we evaluate these parameters with the empirical mean as can be seen in Equation 6

$$mean\_items[p, a] = \frac{mean\_items[p, a] * seen[p, a] + bought[p]}{estimated\_items[p, a]} \tag{5}$$

$$num\_products[p, a] = \frac{1}{mean\_items[p, a]} \tag{6}$$

  where

  - mean_items[p,a] is the mean of the number of products **p** with price **a** bought until now
  - seen[p,a] is the number of times product **p** with price **a** has been bought until the day before
  - bought[p] is the total amount of products **p** bought on the current day
  - estimated_items[p,a] is the number of times products **p** with price **a** have been bought until now (so it is seen[p,a] plus the number of times product p has been bought the current day)

– num_products[p,a] is the inverse of the value computed before because represents the parameter of the Geometric distribution, which we have used to estimate the number of items bought by the customers, once they have decided to buy that product visualized with that price.

• estimate number of products sold (version 2):
Since the number of units bought by a customer depends on the price (so, by the chosen arm), we try to use a UCB-1 algorithm to estimate also this parameter. UCB-1 relies on the assumption that the variable we want to estimate has support in $[0, 1]$, but the number of units bought belongs to $[0, +\infty)$, thus we can not directly estimate the number of units with this algorithm.
One possibility is to directly evaluate the parameter of the Geometric distribution (the inverse of the mean). Normally, UCB-1 uses the upper bound to give an optimistic estimate of the parameters in order to induce the exploration of new arms. However, since increasing the parameter of the Geometric distribution will lead to a decrement in the believed estimation of the number of units bought, we will get a pessimistic estimate. To overcome this issue, we subtract the upper bound from the mean, instead of adding.
The results of this version are significantly worse and are not shown below.

4.1.1 Results

As expected, having less knowledge leads to a solution a little bit worse than the previous step. In other words, the UCB Learner needs more days to converge to the optimal solution. As said before, someday it will change the super arm to pull, in order to explore more.
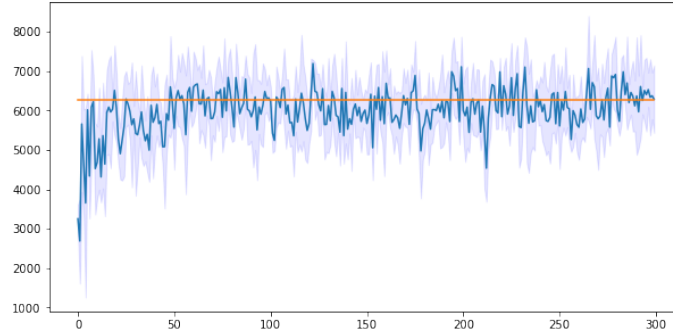
Figure 11: UCB Reward
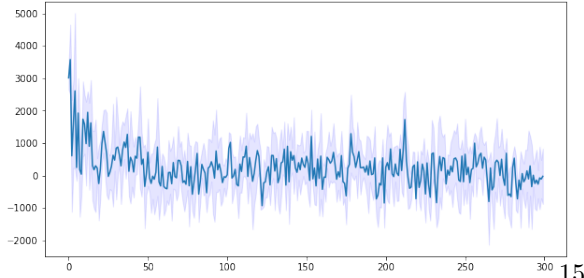
Figure 12: UCB Regret
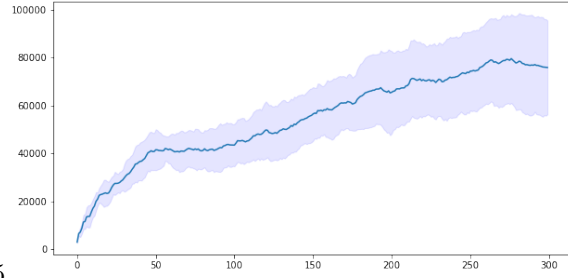
Figure 13: UCB Cumulative regret

15

4.2 TS

Thomson Sampling estimates the alpha ratios exactly as the UCB with the Equation 4.
About the estimation of the number of products sold, the method adopted is the same as the second version for the UCB-1. For obvious reasons, the additional parameters are the α and β parameters of the Beta distribution used for the additional MAB. The update of α is done as follows:

$$\alpha[p, a] = \alpha[p, a] + seen[p]$$

where seen [p] is the total amount of products **p** bought from the first day until now, whereas the update of the β parameters is:

$$\beta[p, a] = \beta[p, a] + tot\_bought[p]$$

where tot_bought [p] is the number of times the product **p** has been bought until now.

The two parameters calculated above are updated in the customer's attributes when the learner has to select which super arm to pull.

4.2.1 Results

The same results as UCB can be observed here, more uncertainty leads to more time to converge. Also this time the results highlight the fact that TS performs better than UCB because it takes less time to converge, generating less cumulative regret.
Furthermore, the reason why using the Thompson Sampling algorithm to estimate both the conversion rate and the number of units sold is more effective than UCB-1 might be because it is less explorative. Having two UCB-1 algorithms may lead to too much exploration and less exploitation.
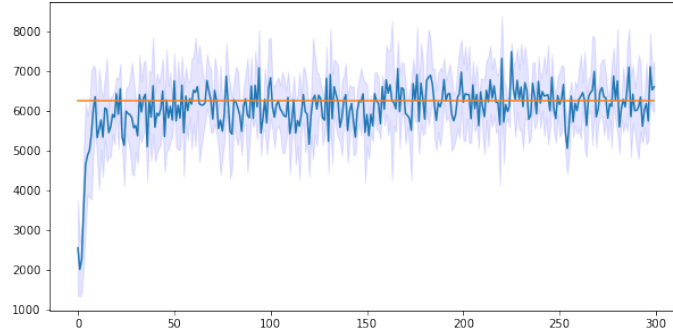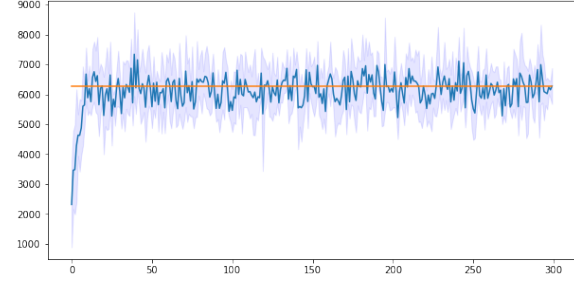

Figure 14: TS Reward
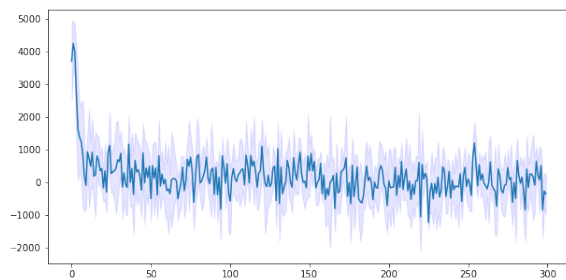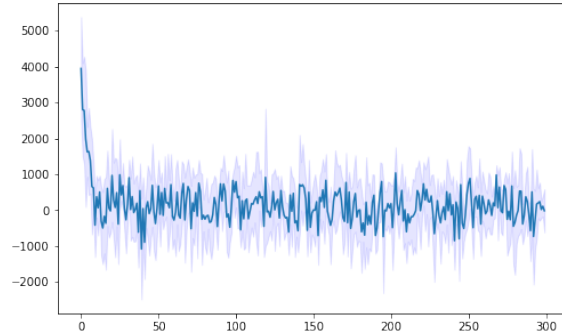
Figure 15: TS Regret
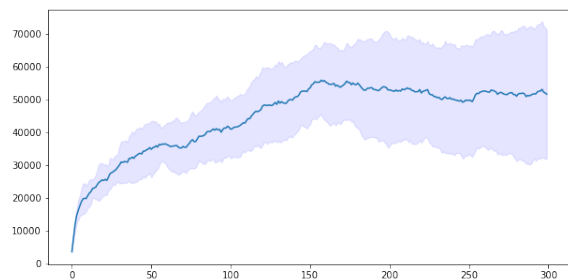
Figure 16: TS Cumulative regret

17

5 Step: Optimization with uncertain graph weights

Consider now the case in which, as in step 3, the conversion rates are unknown but also the values of the graph weights.

These parameters gave us the probability that a customer clicks on the secondary product having a given product as primary.

Therefore, we initialize both learners (UCB-1 and TS) with the graph weights all set to 1.

In the same way as the previous step, below is reported only the additional calculus to estimate the new unknown parameters.

5.1 UCB-1

As said before the first five steps are the same, so here is reported how the graph weights are estimated:

$$estimated\_graph[p,:] = \frac{estimated\_graph[p,:] * bought[p] + clicks[p,:]}{tot\_bought[p]} \tag{7}$$

Where:

- estimated_graph[p,:] is the vector of all the weights of the graph that starts from the node with the product **p**

- bought[p] is the number of times the product **p** has been bought until the day before

- clicks[p,:] is the row of the matrix that reports the number of times an edge of the graph has been activated, which means that a click on the secondary product (columns) has been recorded given a primary product (row). So here we look at all the edge that has been activated starting by the primary product **p**

- tot_bought[p] is the number of times the product **p** has been bought until today (so it is bought[p] plus the number of times product p has been bought today)

5.2 TS

In this case, the Thomson Sampling does exactly the same as the UCB-1 to estimate the graph weights.

5.3   Results

Also in this case the results are very similar to the previous step: both algorithms perform a little bit worse than in the case in which only one parameter was uncertain (conversion rates).
Another thing that remains unchanged is that TS performed better than UCB-1 by reaching the optimal super arm faster.
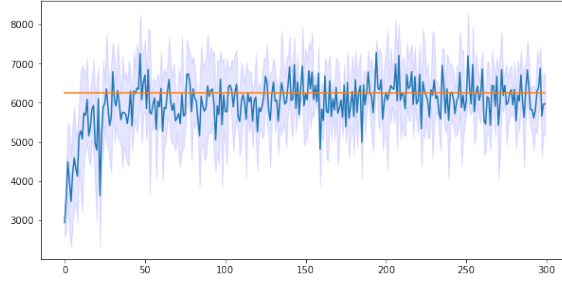
Figure 17: UCB Reward

Figure 18: TS Reward
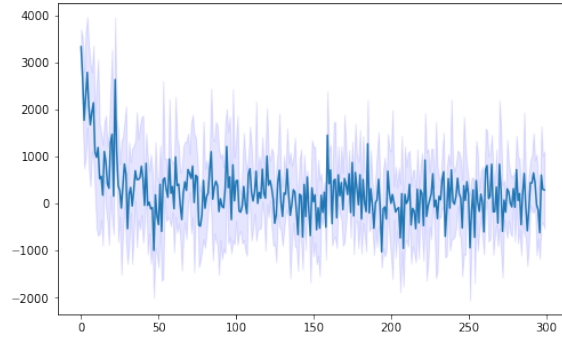
Figure 19: UCB Regret
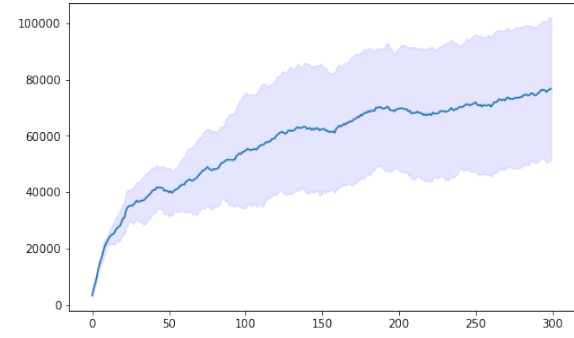
Figure 20: UCB Cumulative regret

Figure 21: TS Regret
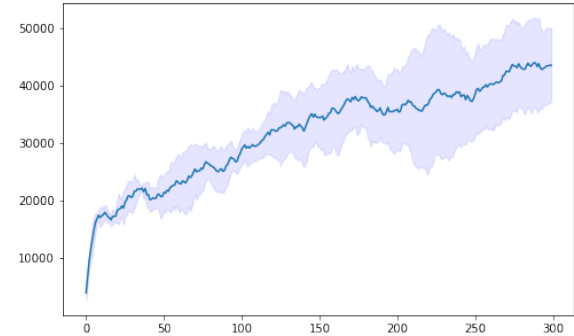
Figure 22: TS Cumulative regret

19

6 Step: Non-stationary demand curve

In this section, we face the case in which the demand curves are unknown and can be subject to abrupt changes. More precisely, our proposed simulator presents an abrupt change every 100 days involving the demand curves for each product for each customer. We use different approaches to face the same problem:

- UCB-1 algorithm with abrupt change detection.

- UCB-1 algorithm with sliding window.

6.1 Abrupt change detection

In this case, at the end of each day, the learner, before updating its arms, checks whether the conversion rate has changed or not for each product for each price. The most naive way is to compute the conversion rate of the current day $\alpha_t(p, a)$ for product p given price a and compare it with the average of the conversion rates of the last days $\hat{\alpha}_{t-1}(p, a)$.
We set a threshold δ such that if :

$$\alpha_t(p, a) \notin [\hat{\alpha}_{t-1}(p, a) - \delta, \hat{\alpha}_{t-1}(p, a) + \delta]$$

then that specific conversion rate has just changed. Due to the high noise that characterizes the realizations of the environment, it is incredibly hard to understand why a realization might be out of the range.
To overcome this issue we propose a slightly advanced approach: instead of comparing the last conversion rate with the average of the last conversion rates, we say that the conversion rate for a specific product for a given price has changed if:

$$\hat{\alpha}_{t,t-\lambda}(p, a) \notin [\hat{\alpha}_{t-\lambda,1}(p, a) - \delta, \hat{\alpha}_{t-\lambda,1}(p, a) + \delta]$$

where $\hat{\alpha}_{t,t-\lambda}(p, a)$ is the average of the last λ conversion rates, whereas $\hat{\alpha}_{t-\lambda,1}(p, a)$ is the average of the first $t - \lambda$ conversion rates for that product p at price a. This method reveals to be more robust to the noise of the environment but requires more days to detect an anomaly, nevertheless, we can achieve great results by setting λ to 5 as we can see from Figures 23, 24, 25.
Once a conversion rate has been spotted as changing, all the data related to that specific pair (price, arm) are dropped since those data are no more relevant.
Here, the code for the detection algorithm:

```python
def change_detection_test(self, pulled_arm, report):
    conv_rate = report.get_conversion_rate()
    for product, arm in enumerate(pulled_arm):
        delta = 0.3
        mean1, mean2 = 0, 0

        if len(self.conv_rate_history[product][arm]) > 10:
            last_conv_rates =
            ↪   self.conv_rate_history[product][arm][-self.splitter:]
            last_conv_rates.append(conv_rate[product])
```

```python
        mean1 =
        ↪   np.mean(self.conv_rate_history[product][arm][:-self.splitter])

        mean2 = np.mean(last_conv_rates)

    if self.t_arms[product, arm] > 11 and (mean1 < mean2 - delta or mean1
    ↪   > mean2 + delta) and not np.isinf(
            self.upper_bounds[product, arm]):
        # detected an abrupt change
        self.t_arms[product, arm] = 0
        self.means[product, arm] = 0
        self.upper_bounds[product, arm] = np.inf
        self.seen[product, arm] = 0
        self.conv_rate_history[product][arm] = [conv_rate[product]]
```

6.2   Insights on abrupt change detection algorithm

Now we want to investigate and understand if, under the assumption to already know all the conversion rates will change, we can improve the performance of the learner: when a change for a single pair (price, product) is detected, all the conversion rates are dropped. From Figures 26, 27, 28 we notice the two algorithms have comparable performances, thus, there is no reason to use a more specific algorithm than the one just introduced.

6.3   Sliding window

Finally, we change the UCB-1 algorithm by implementing the sliding window to keep track of the most relevant samples. The main idea behind this algorithm is the demand curves are currently smoothly changing, thus samples lose importance with time.
In order to avoid old samples to affect the learner's performance, we use a sliding window of size σ: in other words, we compute the estimated conversion rates and their upper bounds using only the last σ samples.
The drawback of the sliding window is the learner has a short memory of the past, thus, if it is used in a stationary environment, it will be outperformed by a stationary UCB-1 algorithm, since it will behave in a more exploratory way. Using a sliding window of size 50, we can see from Figures 29, 30, 31, the learner with the change detection algorithm outperforms the one using the sliding window. This should not surprise us for the following reasons:

- Our environment is characterized by three stationary phases. When a change occurs, it is abrupt instead of being smooth: a sliding window has been thought to face the second scenario, thus it will be slower to converge.

- For the first 100 days, the environment is stationary, thus the sliding window struggles to converge since it is discarding the oldest samples, despite they are still useful.

6.4   Results

In this subsection, we present the experimental results got interacting with a nonstationary environment for 300 days and the number of daily customers is drawn every day from a Normal distribution

$\mathcal{N}(300, 10)$. The experiments are conducted 5 times.
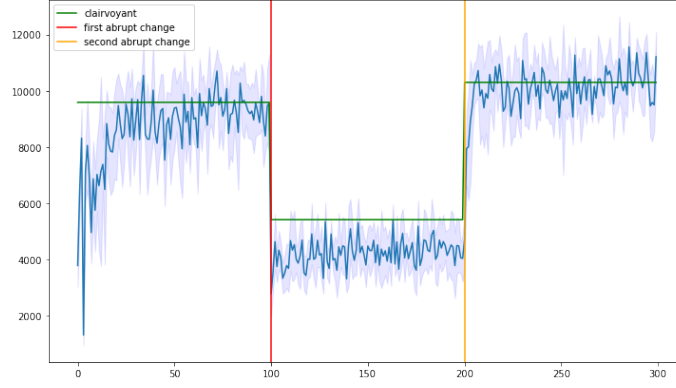
Figure 23: Change detection UCB Reward
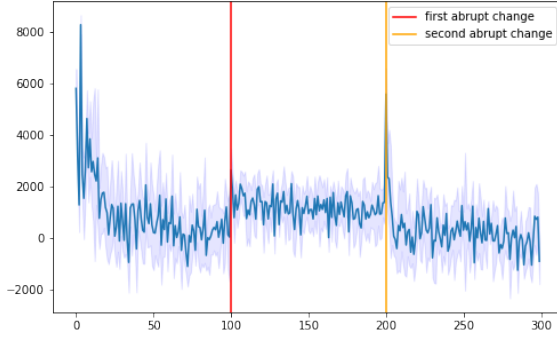
Figure 24: Change detection UCB Regret
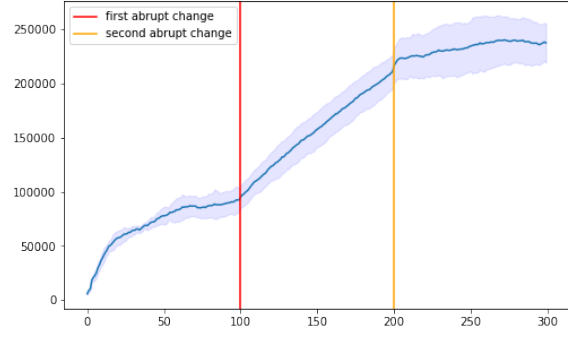
Figure 25: Change detection UCB Cumulative regret
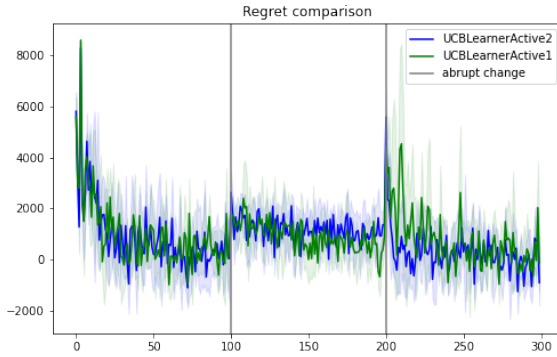
Figure 27: Regret comparison between Learner 1 (algorithm in Subsection 6.2) and Learner 2 (algorithm in Subsection 6.1)
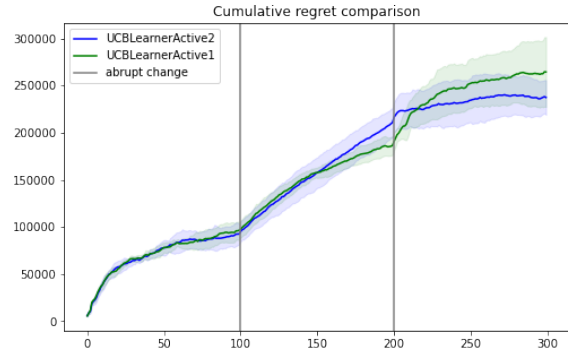
Figure 28: Cumulative regret comparison between Learner 1 (algorithm in Subsection 6.2) and Learner 2 (algorithm in Subsection 6.1)
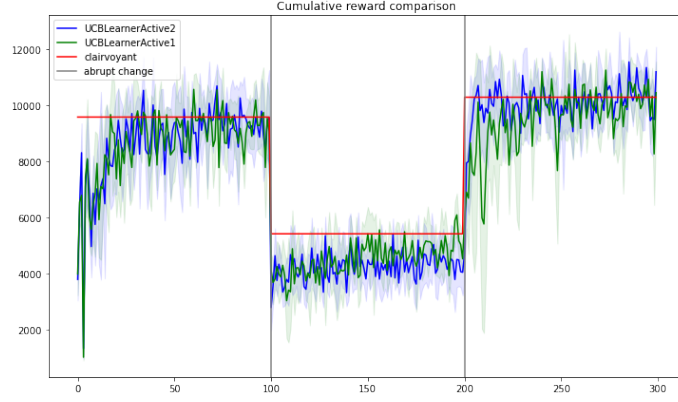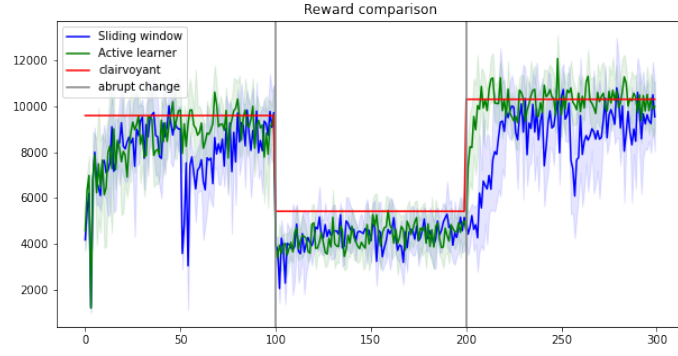
Figure 26: Reward comparison between Learner 1 (algorithm in Subsection 6.2) and Learner 2 (algorithm in Subsection 6.1)

Figure 29: Reward comparison between Learner with change detection algorithm and Learner with sliding window
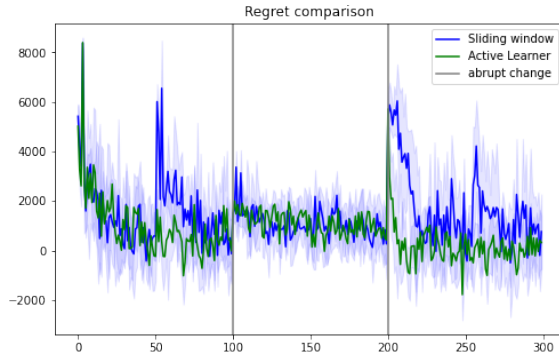
Figure 30: Regret comparison between Learner with change detection algorithm and Learner with sliding window
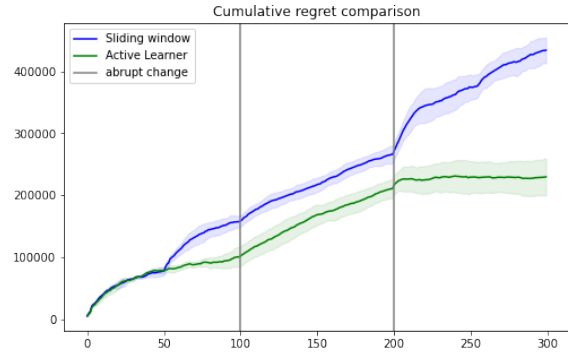
Figure 31: Cumulative regret comparison between Learner with change detection algorithm and Learner with sliding window

7   Step: Context generation

This step is very similar to step 4, however, we can observe the customer features. Therefore, we can set different price levels for every type of customer. To leverage this information, every two weeks (14 days) a context-generation algorithm is executed. The algorithm uses a feature tree to select the learner, and it is built using a greedy splitting technique. Every time the context-generation algorithm is called it generates a completely new feature tree. Consequently, the new tree might be shallower than the previous one.

7.1   How to build the feature tree

Starting from the root node, for each feature the algorithm computes the expected reward without splitting, so by training only a single learner and with splitting on the features, therefore, by training two learners on two partitions of the data. If the lower bound of the expected reward is larger, the node is split in the two cases and the process continues recursively on every new node. The splitting condition is the following:

$$p_l\mu_l + p_r\mu_r \geq \mu, \tag{8}$$

where p_l and p_r are the probability weights and μ, μ_l and μ_r are the expected rewards. The μ value is computed as follows:

$$\mu = \tilde{x} - \sqrt{\frac{5}{|Z|}}, \tag{9}$$

where $\tilde{x}$ is the expected reward computed by simulating the learner and $|Z|$ is the size of the data.

7.2   Results

The results obtained are far from optimal, however, this might be due to various reasons. First of all, the context-generation algorithm used has no guarantees to be optimal, so it is reasonable to be way worse than the clairvoyant. Secondly, the environment is run for a very short period of time, so it might not converge fast enough, not to mention that the environment is very noisy and the high standard deviation of their reward is a direct consequence of that phenomenon. All these factors might be the reasons for the suboptimal results of our approach.

7.2.1 UCB
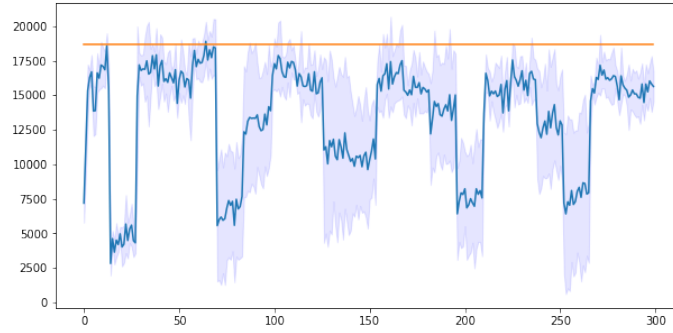
Figure 32: UCB Reward
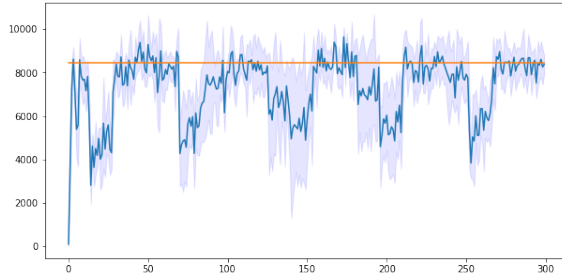
Figure 33: UCB Reward customer 1
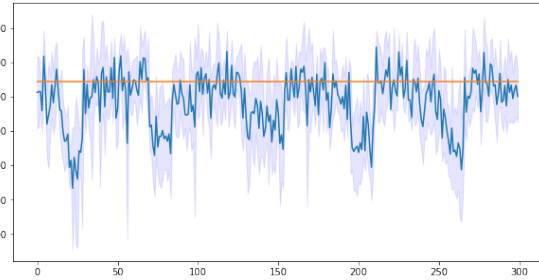
Figure 34: UCB Reward customer 2
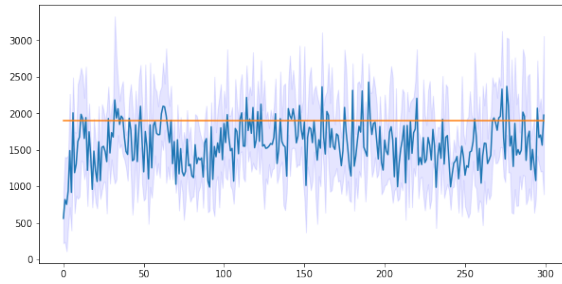
Figure 35: UCB Reward customer 3
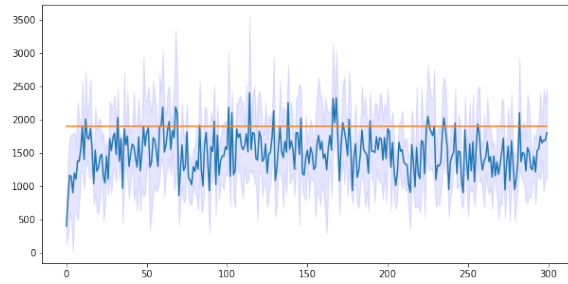
Figure 36: UCB Reward customer 4
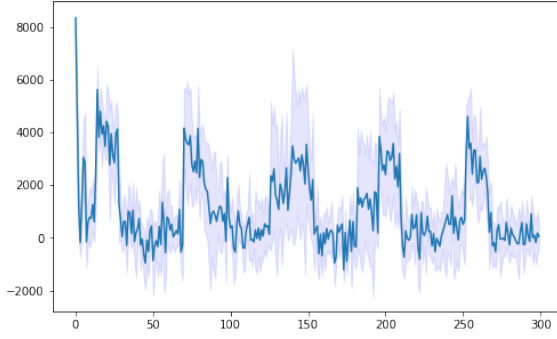
Figure 37: UCB Regret customer 1
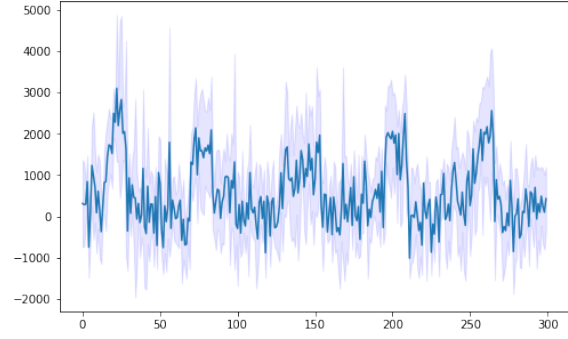
Figure 38: UCB Regret customer 2
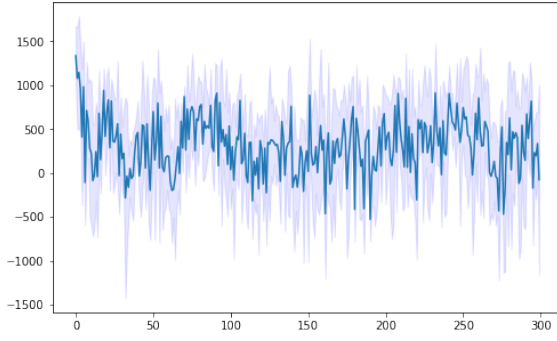
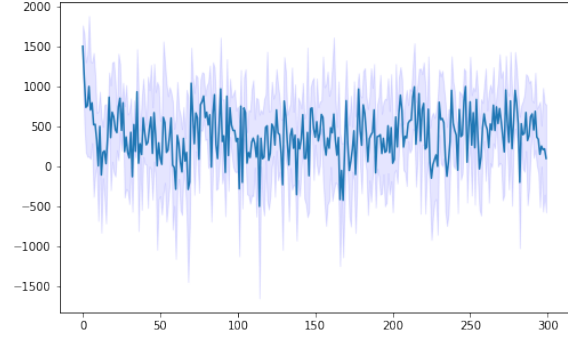Figure 39: UCB Regret customer 3
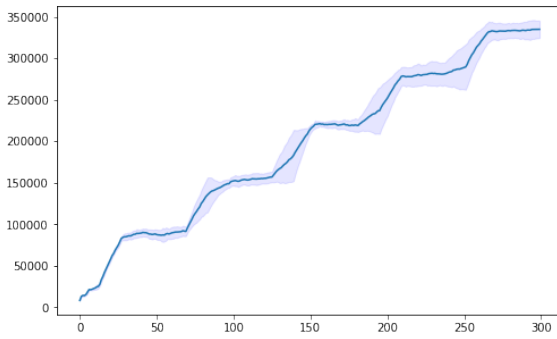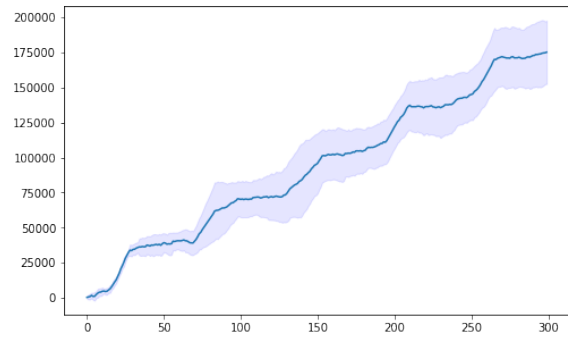

Figure 40: UCB Regret customer 4

Figure 41: UCB Cumulative regret customer 1
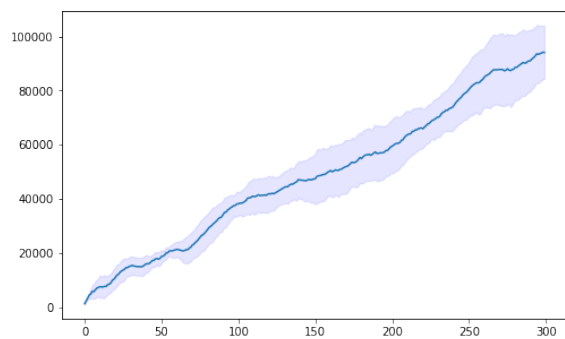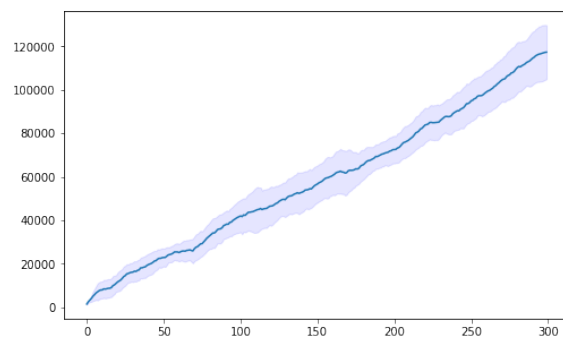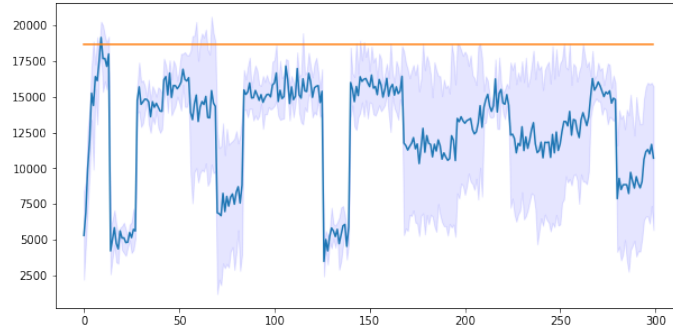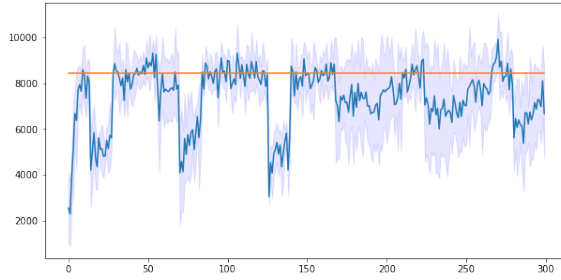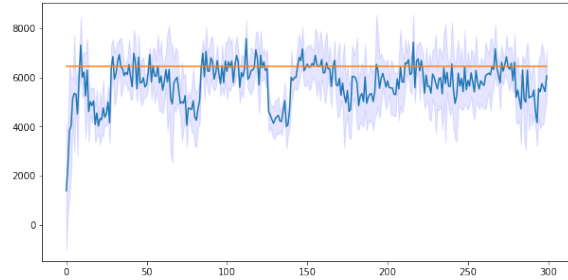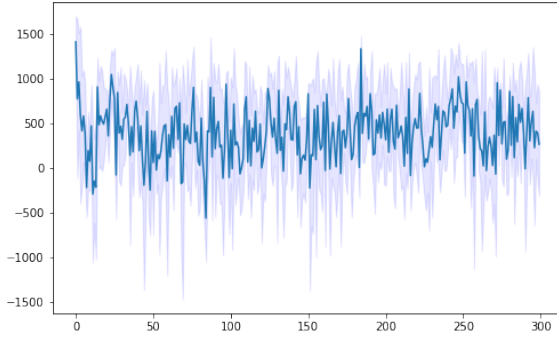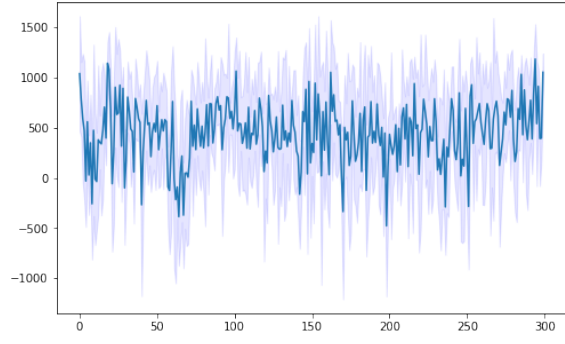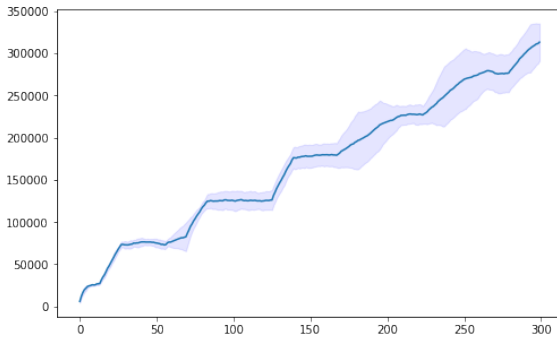

Figure 42: UCB Cumulative regret customer 2

26

Figure 43: UCB Cumulative regret customer 3

Figure 44: UCB Cumulative regret customer 4

7.2.2 TS

Figure 45: TS Reward

Figure 46: TS Reward customer 1

Figure 47: TS Reward customer 2
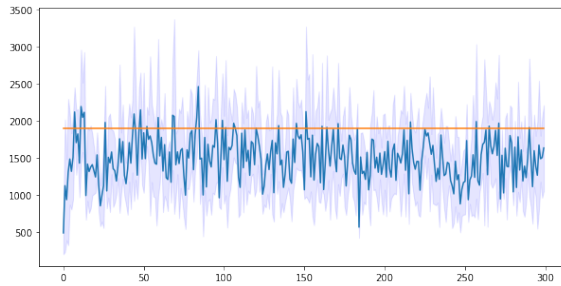
Figure 48: TS Reward customer 3
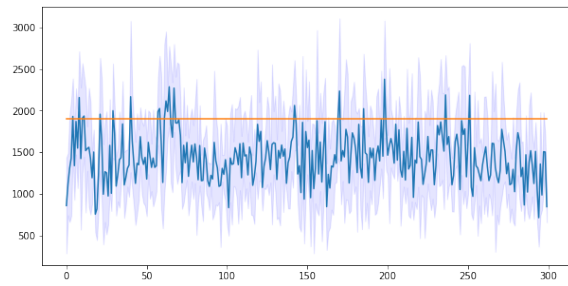
Figure 49: TS Reward customer 4
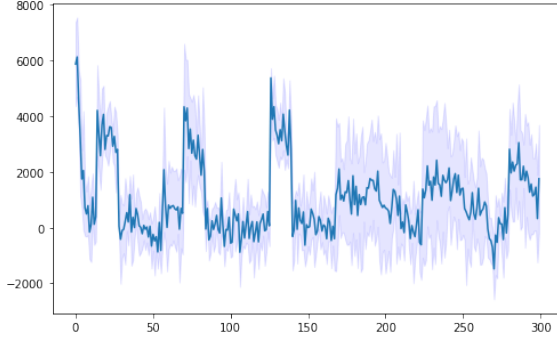
Figure 50: TS Regret customer 1
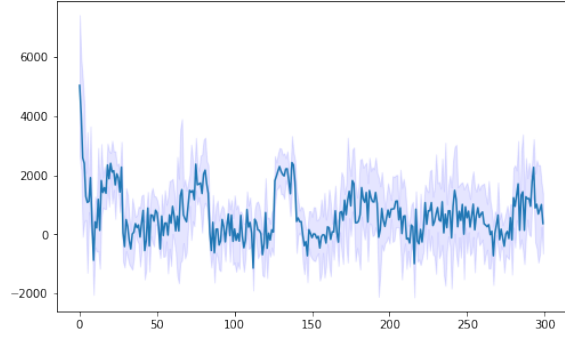
Figure 51: TS Regret customer 2

Figure 52: TS Regret customer 3

Figure 53: TS Regret customer 4

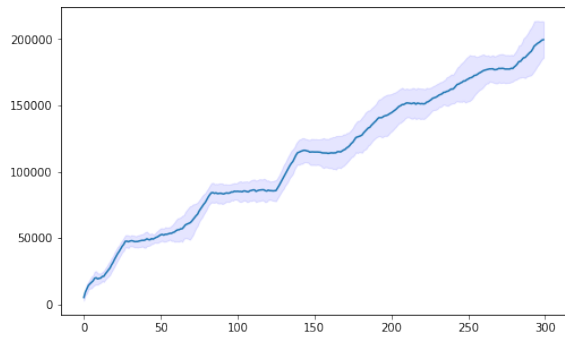Figure 54: TS Cumulative regret customer 1

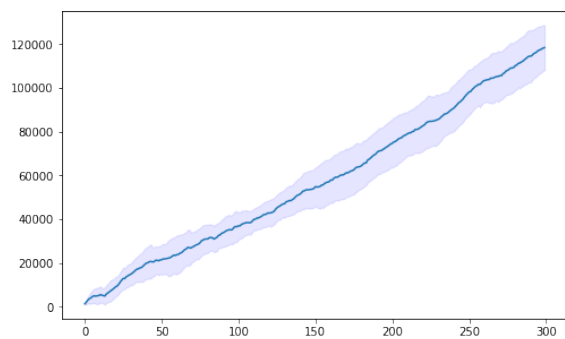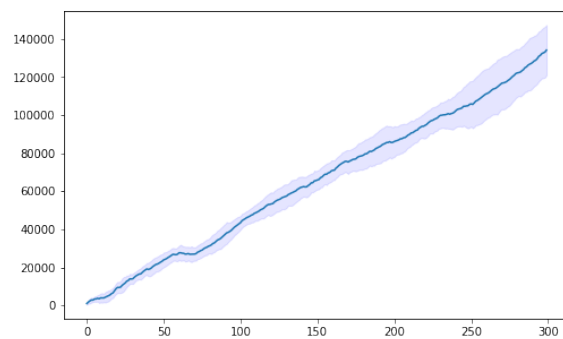Figure 55: TS Cumulative regret customer 2

29

Figure 56: TS Cumulative regret customer 3

Figure 57: TS Cumulative regret customer 4