Project Report - Online Learning Applications
Prof. Nicola Gatti

Social Influence and Pricing

POLITECNICO
MILANO 1863

Caspani Federico [10622658]
Citterio Matteo [10620055]
Cominoli Carlo [10629497]
Cutrupi Lorenzo [10629494]
Panza Loris [10781716]

1 Environment

The scope of the environment is to make a consistent guess on the logic flow of an user who interacts with the company website, in order to train the agent with a coherent approximation of the effective case. In this way the learner will work properly also in real-world scenarios.

1.1 Pricing environment

The real-world scenario to be modeled is the pricing problem of a company that wants to choose what price to sell an item having certain information, and estimating others.
The environment aims to simulate the interaction between a user and the website, and to do so it needs some probability distribution to work from, which were chosen at priori and inserted in a json as following:

- alpha ratios which represent the chance of a user landing in a certain product of the website. The element 0 represents the chance of a user landing on a competitor product, element 1 the chance of landing at product 1 of the website, and so on for all the other products of the company.

```
[0.04, 0.25, 0.29, 0.11, 0.07, 0.21],
```

- prices which represent the different amount to be tested. Each row is related to a product and the number of prices for each product is four. For the sake of simplicity in our problem price and margin are equivalent, which is possible in case of products that have no producing costs such as lectures etc...

```
[5, 6.5, 8, 10],
[16, 17.3, 18.5, 19.3],
[67, 69.8, 72, 73.6],
[12, 13.5, 15.7, 16.2],
[101.4, 102.3, 103.5, 105]
```

- conversion rates which represent the probability of a user to buy a product once he landed on the page of it. In the document each row is related to a product and each column to a price (element [0,0] represents the probability of the user buying the first product at the first price)

```
[0.9, 0.45, 0.4, 0.35]
[0.4, 0.8, 0.3, 0.25],
[0.5, 0.45, 0.9, 0.35]
[0.2, 0.18, 0.8, 0.1],
[0.5, 0.45, 0.4, 0.9]
```

- secondary products which are the products suggested to the user once he buys the primary product. In our problem every product has two secondaries which are fixed.

```
[[1,2],[0,2],[0,1],[0,1],[0,1]]
```

- number of product sold which represent the mean amount of items that a user purchases once he decides to buy (element [0,0] states that user purchases in mean 3 items of the first product at the first price, once he decides to buy it)

Furthermore the users are classified in different classes based on two boolean features, where each class is characterized by a unique combination of features and unique informations (each class has different alpha ratios, conversion rates, and number of items bought at every purchase), simulating different types of users that act differently from users of different classes but similarly to the ones belonging to the same class.

```
[3, 2.5, 2, 1.5],
[5, 4.5, 3, 2],
[3, 2.8, 2.6, 1.5],
[4, 3.8, 3.6, 2.5],
[2, 1.9, 1.8, 1.75]
```

1.2 Social Influence

In order to properly work, any pricing algorithm needs to interface with the environment and obtain some meaningful information about the results of the pulled arm.
To simulate the behavior of the users, from which these information derive, some social influence techniques are resorted. In particular the main algorithm that used is Monte Carlo Sampling.

1.2.1 Simulation of user interaction

In the problem case it is required to simulate the behavior of the user to estimate his probability of seeing a certain product and his probability of buying that specific product at that specific price, once he has seen it.
In order to do so the classes below are implemented:

- The Graph class: As specified on the assignment, the site can be expressed like a graph where the nodes represent the products of the e-commers and the edges the probability of seeing a product starting from another directly connected one.
- The Page class: it represents an html page opened by the related customer and it contains the primary, secondary and third product displayed on the page.
- The Customer class: it represents a single customer landing on our e-commers site.
  In particular this class maintain two important arrays:
  - Pages: contains a list of active pages.
  - Products state: every element shows the state of each product (inactive, susceptible, active).
- The Simulator class: This class is the one that implements the Monte Carlo algorithm in our problem.

The simulator class works as follow:
Every simulation is about a single customer and it simulates his behavior until he closes all the opened pages and finishes the interaction with the website.
It assigns to the customer the first page visited with a probability coherent with the alpha ratios.
In doing so the state of that product is updated as active in the list of states of the customer.
Then, until there is at least one page still opened:

- With a random probability influenced by the conversion rate of the primary product of the selected page he buys or not the product.
  - If the choice is to buy the product then it is simulated a bought quantity and it's chosen if the customer will visits one of the secondary products or it will close the page accordingly with the edge probabilities.
    * If the choice is to visit some other product (that has to be susceptible for the customer), then a new page is generated for the product and added to the list of active ones.
      Also the state of the product for the customer is updated as active.
    * if the choice is to not buy, the page is removed from the customer list of active pages and the state of the primary product of the closed page is updated as inactive for the customer. Then the loop will restart and the simulator will take in consideration another page if at least one is left, else it will end.

    At the end of the above interactions, the page is removed from the list of the active pages.
  - if the choice is not to buy then the page is removed from the list of active pages in the customer and it is set the primary product of the closed page as inactive for the customer. Then the loop will restart and the simulator will take in consideration another page if at least one is left, else it will end.

At the end of the simulation the data about which product the customer has visited and which and how many product he has bought is returned to the learner.

With this approach it is possible make a consistent simulation of a user interaction and be sure that the information gathered respect the real problem.

1.2.2 Monte Carlo

Monte Carlo is a reinforcement learning technique that allows to learn the model of the environment by interacting with it and then evaluating the information with a frequentist approach. The Monte Carlo algorithm can be expressed in this way:

1. For every node i assign $z_i = 0$.
2. Generate randomly a live-edge graph, according to the probability of every edge.
3. For every node check if it is active in the given live-edge graph and in case assign $z_i = z_i + 1$.
4. Go to step 1 unless k repetitions have been done.
5. For every node return z_i/k.

In the problem scenario, this method is used to estimate the probabilities of the edges of the graph and the probabilities to buy each product, since running the simulation of user interaction a sufficient number of times gives a good approximation of them.

1.3 Assumptions for the environment

In order to have a solution which is clearly better than all the other possibilities, all the products have a price whose conversion rate is way better than all the other prices of the same item. Other parameters have been simplified too: margin and price are equal, alpha_0 (the chance of a user ending in a competitor page) is fixed to 0, λ (which represents the probability of an user opening the tertiary product with respect to the secondary) is fixed to 1, and secondary products are chosen in order to have a shallow graph navigation.

1.4 Nearby reward

One of the main problems of this scenario is the calculus of the indirect reward generated by each product: indeed, since buying a primary product leads to a probability to open a secondary and, eventually, to buy it, every price of every product generates a different nearby reward that needs to be estimated to better understand which is the combination of prices that leads to the highest profit. But this calculation is not trivial, since there are many factors that influence it, such as the different conversion rate that each price has, or the impossibility for a user to open a product he already visited.
So, finding a way to properly estimate it is a priority.
An exact calculation requires a too complex model (the authors sketched a method that required a 4-dimensional and was really hard to implement it, so it was discarded and not included in the code) so it is required another approach which is more practical and easier to implement, and Monte Carlo Simulation can be a good solution and it is the one used in the problem. For every pair [product,price] a Monte Carlo simulation like the one described in the previous paragraph is performed, and at the end of all the iterations the probability to visit each product starting from a primary one is calculated doing the rateo between the times the node referring to the secondary product has been visited and the number of iterations

```
    def simulateTotalNearby(self, selected_price):
        #matrix where each row refers to a primary product, each
        #column to a secondary
        times_visited_from_starting_node = np.zeros((self.n_products,
                                                     self.n_products))
        for prod in range(self.n_products):
            for _ in range(total_iterations):
                #visited_products contains the products that have been visited
                #starting by primary product prod at this iteration
                visited_products_ = self.simulateSingleNearby(selected_price,
                                                              prod)
                for j in range(len(visited_products_)):
                    #updates every secondary visited at this iteration
```

```
                if (visited_products_[j] == 1) and j != prod:
                    times_visited_from_starting_node[prod][j] += 1
        return times_visited_from_starting_node / total_iterations
```

As stated above, the method performing an user interaction (here called simulateSingleNearby) is similar to the one used in the environment, with the difference that the only object to return is a list of the visited products, and that some parameters used are just estimations of values and not real ones (for example in step3, since the learner doesn't know the real conversion rate but aims to estimate it, this estimation is used to simulate the conversion rate in the method simulateSingleNearby). The Monte Carlo Simulation is a great tool for this scenario since it solves a problem of complexity without losing much of accuracy, and saving a lot of time depending on the number of iterations performed.

1.4.1 Number of iterations

Having an amount of iterations that is big enough to do a good estimation but short enough to have low time complexity is essential, and in this regard a theorem comes in help, which states

Theorem 1 *With probability 1 - δ the estimated activation probability of every node is subject to an additive error of $\pm\epsilon n$ when the number of repetitions is equal to*

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

$$R = O((1/\epsilon^2) log(|S|) log(1/\delta))$$

where n is the number of nodes and S is the number of seeds

Since in this problem both n and S are equal to the five product to study, to have an error of 10% with probability of 95%, δ is set to 0.05 and ϵ to 0.1÷5=0.02. So the amount of repetitions to perform is equal to

$$R = O\Big(\frac{1}{0.01^2} log(5) log\frac{1}{0.05}\Big) = 2274$$

$$R = O((1/0.01^2) log(5) log(1/0.05)) = 2274$$

so 2274÷5=455 iterations for each product

1.4.2 Considerations

The nearby reward encourages the prices with the highest conversion rates since they allow the graph to be activated more and generate more indirect traffic. This means that it could happen that sometimes a price with a very low margin of profit but high conversion rate is preferred to a price with an higher margin but lower conversion rate, especially when the difference between the two margin is very low compared to a big difference in the probability of an item to be sold.

2 Greedy Optimization Algorithm

The goal of the problem is to maximize the total cumulative expected margin of the products knowing all the informations of the environment, and to do so it is implemented a greedy algorithm that works as following:

1. starts from the lowest price for every product as the best configuration
2. evaluates the margin obtained by increasing the price of one of the five products
3. selects the configuration with the highest value between those calculated at the previous step and the current best configuration
4. repeats from step 2 with the one selected in the previous step as the best, until no new configuration between those just evaluated is better than the previous best

Clearly, it is important to implement a simple method that given a combination of prices, returns its expected cumulative reward

```
    def revenue_given_arms(self, arms):
        revenue = 0
        for i in range(self.n_products):
            revenue += (self.prices[i][arms[i]]
                        * self.conversion_rates[chosen_class][i][arms[i]]
                        * self.num_product_sold[chosen_class][i][arms[i]])
        #Calculations regarding nearby reward are ousted
        return revenue + nearby_reward
```

where arms is the input combination of prices, and the revenue is calculated as the sum of the multiplication, for every item, of the conversion rate of a product at a certain price (given from arms), the price of selling the product at that price and the mean number of product sold for purchase at that price.

2.1 Limitations

This approach has some evident limitations in the exploration phase, so it will generally not converge to the optimal combination of arms to select. To better understand the meaning of limitation in exploration, let's face an example in which the best combination of arms is 2-2-2-2-2. The algorithm will start from 0-0-0-0-0, but if there is no improvement in the cumulative reward by increasing the price of one of the products, the algorithm will stop at the first iteration, without ever trying the best combination. This approach can be used when the number of possible arms is so high that even trying all the possible combinations once is impossible in practice, so a method that will iteratively compare a limited number of possibilities is preferred, and it is even better when there is a previous knowledge that suggests what combination to start from. However the problem faced in this report is not the most suited for this algorithm, since the number of combinations is not so high and doing a proper exploration is the best choice to find what price to sell each product.

2.2 Results

As stated above the results are not correct and the algorithm stops before even trying the best combination of prices, meaning that the average reward is smaller than the clairvoyant one and the cumulative regret will keep increasing.

```
Greedy algorithm chosen arms:  [[0, 1, 0, 2, 0], [0, 0, 1, 0, 0], [1, 0, 1, 1, 1]]
Clearvoyant best arms: [[0, 1, 2, 2, 3], [0, 2, 1, 0, 2], [1, 3, 1, 1, 1]]
Average reward with greedy algorithm choices:  95.62666912087911 108.16039914835166 149.66023357211537
Average reward with best arms:  143.5094723076923 147.1028799664835 180.20642418956044
Average regret per iteration:  48.548290054945085 41.9687147423077 35.348263581730805
```

3 Optimization with uncertain conversion rates

Differently to the section above, the probability of an user to buy an item once he visits the item's page is now unknown and needs to be estimated. This is common in real-world scenario where the amount of people buying a product can't be known at priori. To properly estimate these probabilities the most famous Multi Armed Bandit algorithms are used: UCB-1 and Thompson Sampling. Both these algorithms work using arms (which in this case represent the prices of each product) which are selected at the start of the day, and based on the arms chosen the environment gives a reward (which in this case indicates if a user who ended up in the webpage of a product effectively purchased it) and the algorithms update their informations based on the result obtained. In this problem the arms are selected at the beginning of the day while the informations are updated at the end of the day. For the sake of simplicity, the number of daily iterations (amount of visitors to the products to study) is fixed to 50, as well as the number of days of the study, which is 60.

3.1 UCB-1

Upper Confidence Bound 1 is a deterministic algorithm that usually works in this way:

1. Initially fixes the mean probability and the bound length of every price of every product respectively to 0 and +infinite

2. Computes the probability mean of a product visited to be sold at a certain price

$$m = \frac{\#NumberOfSuccesses}{\#NumberOfTimesPulled}$$

3. Computes the bound length of each price of each product

$$w = \sqrt{\frac{2logT}{\#NumberOfTimesPulled}}$$

4. Plays the arm with the highest m + w

$$arm = \underset{a}{\operatorname{argmax}}\, m(a) + w(a)$$

5. Observes the reward given by the environment and updates its parameters accordingly

6. Repeats from step2 until the simulation ends

At the start of the day the most promising combination of arms is chosen, meaning that for each product the most promising price must be chosen. Obviously in this problem not only the probability of an item to be sold is the criterion to choose it, but the price that gives the highest expected reward, which depends on price of selling, average number of product sold, probability to sell and nearby reward, so step 4 of the algorithm is changed to:

```
    def act(self):
        idx = np.argmax((self.widths + self.means) *
                        ((self.prices*self.num_product_sold)
                            + self.nearbyReward), axis=1)
        return idx
```

In this problem optimization the average number of product sold for each price is known, while the other parameters of the formula aren't, so they must be estimated, and as specified previously, this operation is performed at the end of the day

```
    def update(self, arms):
        self.nearbyReward = updateNearbyReward(arms)
        self.means = updateMeans()
        self.widths = updateWidths()
```

In order to properly update all the parameters there are some lists used to store all the history of the algorithm (for example there is a list that for each combination of product and price keeps track of the amount of successes and failures of it, so it can be used to estimate means and widths).

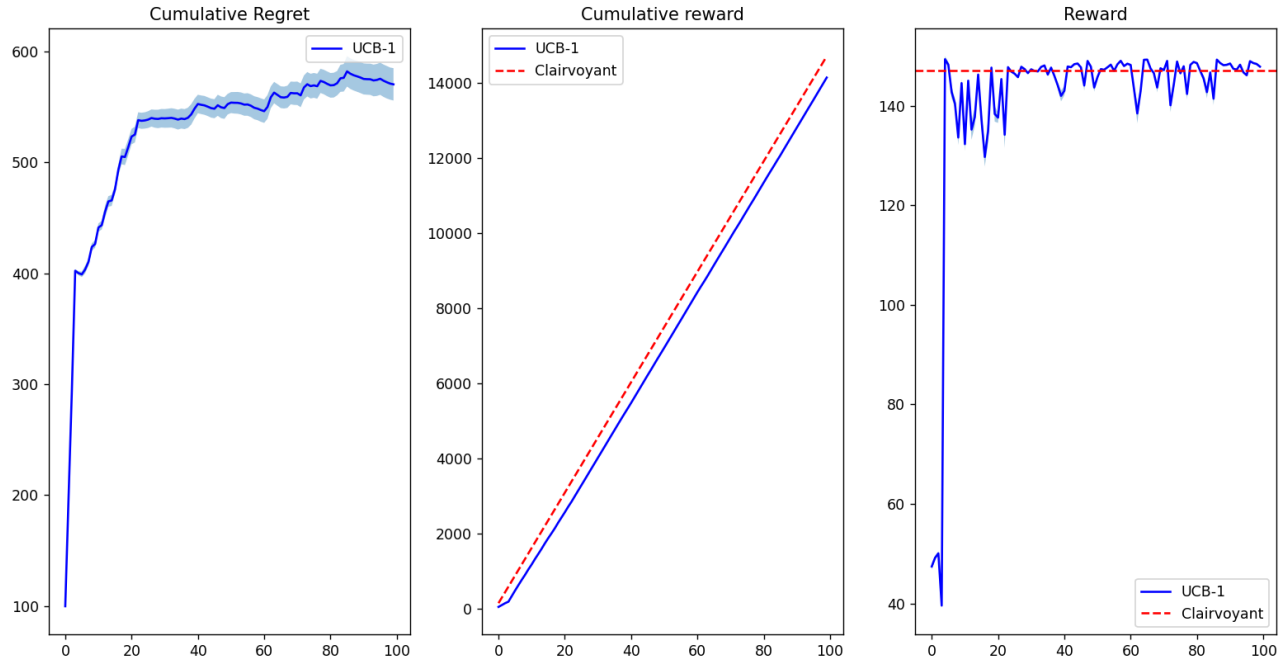
3.1.1 Results

In an environment where some arms are way better than others, like the one used in this problem, the algorithm converges almost immediately.

```
[0 0 0 0 0]
[1 1 1 1 1]
[2 2 2 2 2]
[3 3 3 3 3]
[0 1 2 2 3]
[0 1 2 2 3]
[0 1 1 2 3]
```

In facts, when the number of days is fixed to 100, daily iterations to 500 and the number of plot iterations to 5, the pure-exploration phase lasts only four days, where the algorithms tries every possible price for every product, but then it already understands what's the best combination of prices and mantains it, although at some days it might try some new combination, maybe changing the price of one product. This behaviour happens because arms that are not pulled by a long time have a big bound length, which is crucial in the arm

choice, due to the formula of step 3, so from time to time prices that were not tested by a long time are tried again.



A quick convergence determines a small regret since the best prices are already chosen by the first days of the study. In the image it can be seen that almost all the regret is related to the first exploration phase, while later on the algorithm tends to proceed linearly at the same speed of the clairvoyant. This means that in this scenario and using a simple environment the algorithm works great.

3.2 Thompson Sampling

The TS algorithm works similarly to the UCB-1 but instead of computing the sum of the mean value and the bound length for each arm, it samples a value for every pair [product, price] based on a probability distribution, and picks the highest sample for every product. This process is indeed stochastic, since the most promising arm is not always the one which is executed, and the probability distribution is drawn according to parameters called Beta-parameters, which in facts contain the number of successes and failures for every arm. Updating correctly these parameters is crucial for the correct working of the algorithm, and like the UCB-1, this operation is performed at the end of the day. The first element of the beta parameters, called alpha, contains the amount of successes of a certain price of a certain product, while the second element, called beta, contains the amount of failures of the pair. Trivially the number of times an arm is pulled is alpha + beta.

```
    def update_beta_distributions(self):
        self.beta_parameters[:, :, 0] = self.beta_parameters[:, :, 0]
                                        + self.success_per_arm_batch[:, :]
        self.beta_parameters[:, :, 1] = self.beta_parameters[:, :,1]
                                        + self.pulled_per_arm_batch
                                        - self.success_per_arm_batch
        self.pulled_per_arm_batch = np.zeros((self.n_products, self.n_arms))
        self.success_per_arm_batch = np.zeros((self.n_products, self.n_arms))
```

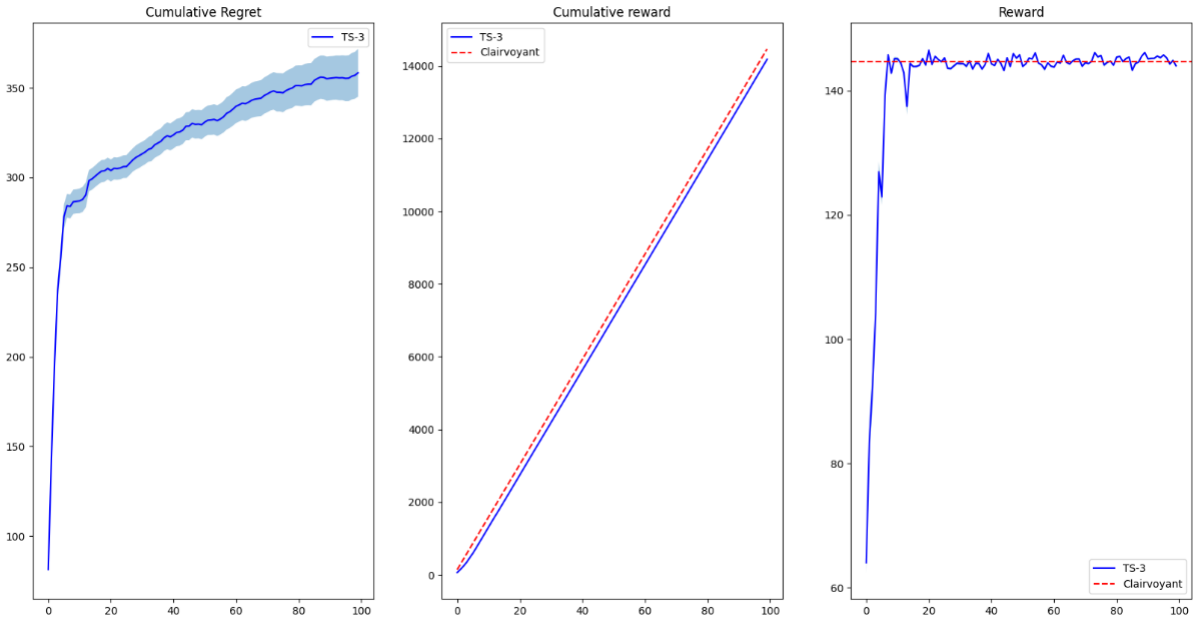
Pulled_per_arm_batch and success_per_arm_batch are the data structures containing the daily informations of every arm, so they can be used at the end of the day to update the values of the total successes and insuccesses of each pair [product, price]. With these beta parameters, it is not required to calculate means and widths like in the UCB, because samples are drawn according to the values of alpha and beta:

```
    def act(self):
        idx = [0 for _ in range(self.n_products)]
        for prod in range(self.n_products):
            #generate beta distribution for every price of the current product
            beta = np.random.beta(self.beta_parameters[prod, :, 0],
                                  self.beta_parameters[prod, :, 1])
            #updates the arm of the current product with highest expected reward
```

```
            idx[prod] = np.argmax(beta * ((self.prices[prod] *
                        self.num_product_sold[prod]) + self.nearbyReward[prod]))
        return idx
```

3.2.1 Results

In this scenario Thompson Sampling algorithm converges fast to the optimal combination of arms so the initial regret is quite small, and later on the clairvoyant cumulative reward and the algorithm's one tend to be equal.



This means that Thompson Sampling is a great approach in this situation and the loss due to the exploration phase is almost zero. This is due to the fact that TS works best in situations where there are some prior informations that influence the arm choice, like in this case where most of the parameters are known (such as the number of items sold per product or the probability to click on a secondary product).

3.3 Comparison

Both the algorithm have good result in this simple environment, despite UCB-1 has a deterministic approach and Thompson Sampling a stochastic one. TS has overall a lower regret since it converges faster than UCB to the optimal arms, due to the fact that instead of doing a pure exploration phase like in the UCB (where each arm is chosen at least one time) it will keep choosing arms which got good results in the first iterations. Moreover Thompson Sampling algorithm is more consistent in the selection of the best arms at the start of each day, while UCB eventually chooses arms which have not been selected by a long time. This difference results in better performances for TS when the environment is static like in this case, because it will exploit more the best combination, while UCB could try arms that had bad results before, but could potentially work better in the current situation.

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

In this scenario there is more uncertainty, indeed also the α-ratios and the number of items sold are unknown to the learner, as well as the conversion rates. This is also a real-world scenario since usually the learner doesn't know what's the average of items sold per iteration neither the probability of a user to end in a certain page, so it has to estimate these parameters at run-time. Actually only the number of items sold per product is crucial for the problem to solve, while the α-ratios are not useful and can be ignored, but they are anyway computed to provide a better graphical representation of the results. Estimating α-ratios is not useful in the optimization itself since the problem to solve is finding the best price for each product, but the probability for an user to end up in a product's webpage as first product is not influenced by the price chosen or the conversion rate of such

price, but it is identical for every price of the same product. This means that they can be ignored for the scope of this optimization.

4.1 Implemented algorithms

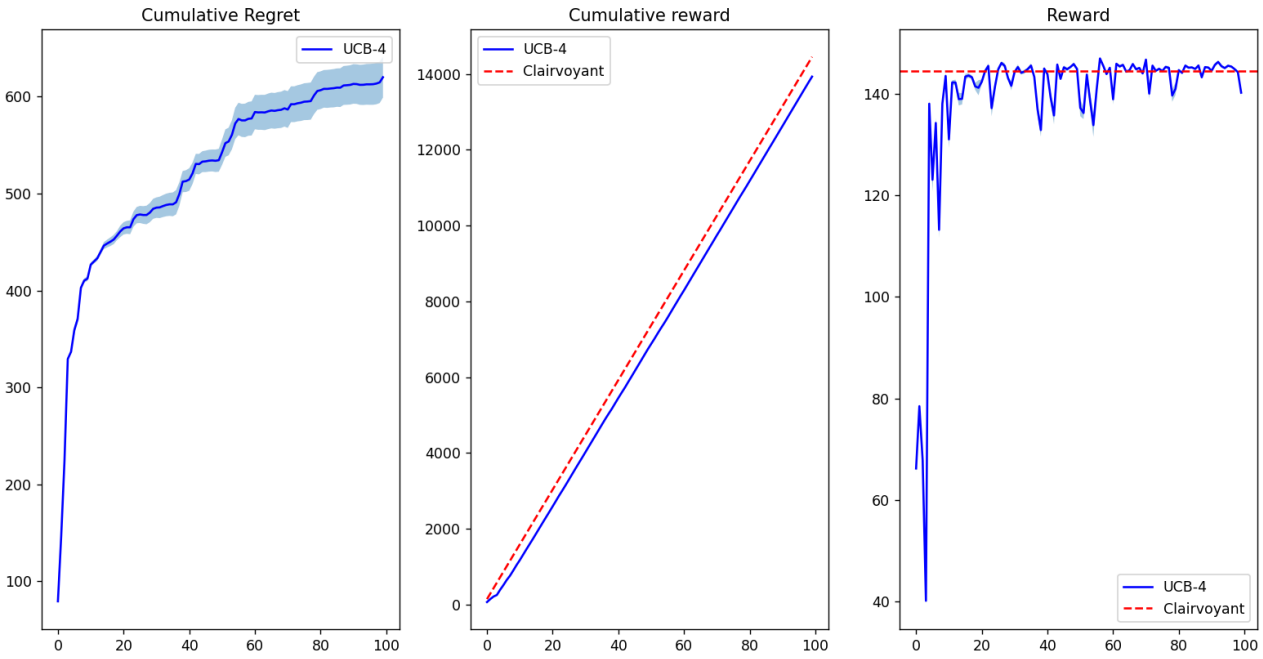
The algorithm used for this purpose are the same of the step before, UCB-1 and Thompson Sampling, with the difference that now the number of items sold per product are unknown and need to be estimated. For this reason a new data array is created, which contains the amount of items bought by an user once he decides to purchase the product, and it is used to properly estimate the average of purchases, which are stored in another array. This array is initialized to all ones values since it is known for sure that once a user decides to purchase a product, he will buy at least one item, and will eventually increase according to the new data collected. Initializing this number to infinite could also be a good idea, since it enhances an initial exploration of all the prices.

```
    def estimate_num_product_sold(self):
        for prod in range(self.n_products):
            self.num_product_sold_estimation[prod][arm_pulled[prod]] =
                        np.mean(self.boughts_per_arm[prod][arm_pulled[prod]])
```

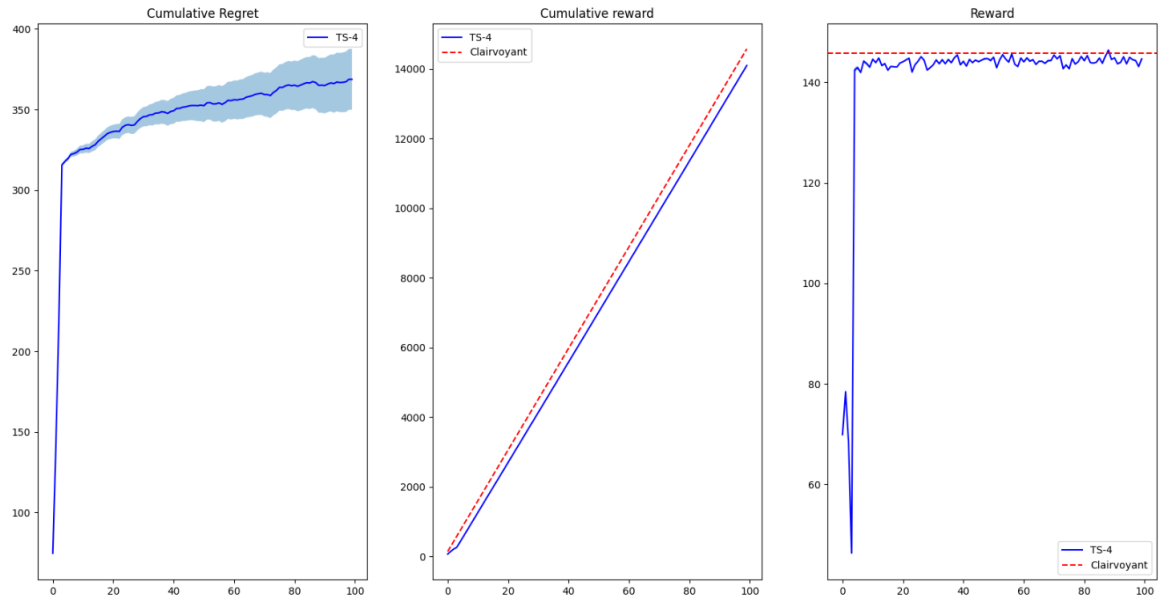
The other difference of the algorithms with respect to the old ones regards the computation of the nearby reward, which now depends on the estimation of the number of product sold instead of the real value, meaning that also the estimation of the nearby reward will be a little less precise.

4.1.1 Results

Having less knowledge obviously means more approximation, which leads to a slower convergence to the optimal solution, due to the fact that with a little amount of samples there are more fluctuations in the number of items sold per product estimation, especially for those products that have a low conversion rate, since there are very few samples of items sold at the first iterations, which means that the algorithm requires a longer exploration phase to have a good enough approximation of the amount of items sold per product. In the end the algorithms are overall slightly worse than the previous, due to the higher approximation, but will converge to the best combination of prices if given a big enough amount of samples.
When the study parameters are the same as the point 3, UCB-1 requires between 10 and 20 days to converge to the optimal arms, meaning that the starting regret will be larger, and overall the algorithm has a smoother curve regarding cumulative regret, which anyway increases slower by the time, so this method is still a viable option in the new scenario.



With respect to UCB the TS method converges faster to the optimal arms (less then 10 days), for this reason the cumulative regret is lower since the initial regret is smaller.



5 Optimization with uncertain graph weights

In the new scenario the graph weights are unknown, meaning that the learner doesn't know the probability of an user ending in the secondary products. This means that a new parameter must be estimated and to do so, new lists are added, in order to store the amount of time each product has been visited as first primary product and the amount of time each product has been visited as secondary with respect to each product as primary. Although these data structure are complicated, the optimization phase is faster since the estimation of the nearby reward is not based on Monte Carlo Simulation anymore, but on a simple division between two matrices.

5.1 Implemented algorithms

The algorithms implemented are the same of the previous steps, UCB-1 and Thompson Sampling, with the only difference of the addition of new data structures that need to be properly updated, and a new formula to estimate nearby reward.

```
    def updateHistory(self, arm_pulled, visited_products,
                      num_bought_products, num_primary):
        # ... same update part of the previous algorithms ...
        # the new data structures are properly updated at every iteration
        self.times_visited_as_first_node[num_primary]
                                [arm_pulled[num_primary]] += 1
        self.times_product_visited_as_first_node[num_primary] += 1
        if num_bought_products[num_primary] > 0:
            self.times_bought_as_first_node[num_primary]
                                [arm_pulled[num_primary]] += 1
        for i in range(len(visited_products)):
            if (visited_products[i] == 1) and i != num_primary:
                self.times_visited_from_starting_node[num_primary]
                                [arm_pulled[num_primary]][i] += 1

    def update(self, arm_pulled):
        # ... same update of previous algorithms ...
        # the visit probability of each secondary starting from a primary
        # is calculated with the lists updated in updateHistory()
        for t1 in range(self.n_arms):
```

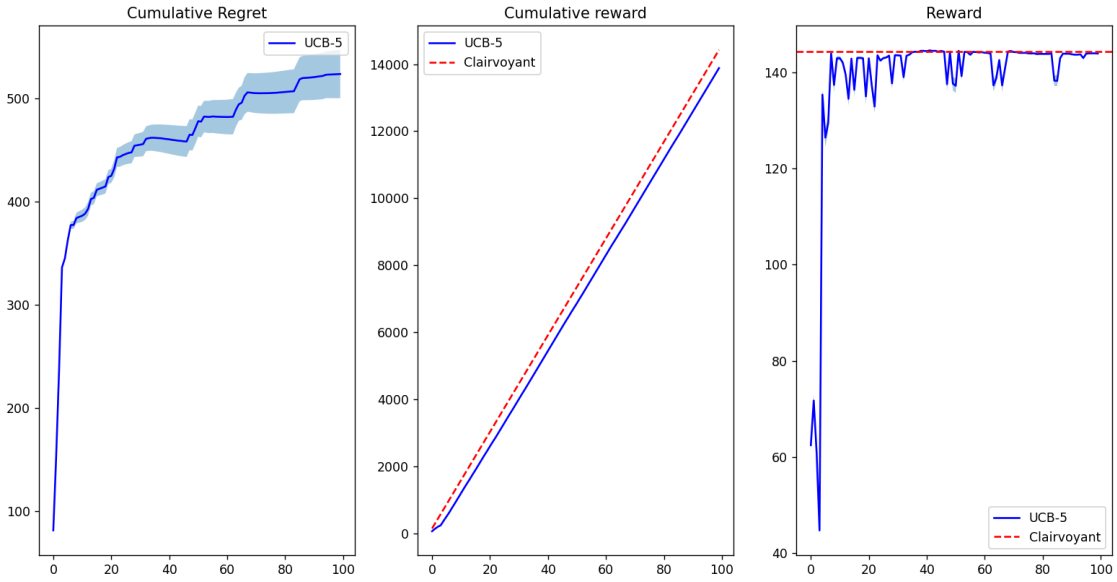
```
            for t2 in range(num_products):
                if self.times_bought_as_first_node[prod][t1][t2] > 0:
                    self.visit_probability_estimation[prod][t1][t2] =
                        self.times_visited_from_starting_node[prod][t1][t2]
                        / self.times_bought_as_first_node[prod][t1][t2]
                else:
                    self.visit_probability_estimation[prod][t1][t2] = 0

    def totalNearbyRewardEstimation(self):
        #returns a matrix containing the nearby rewards for all products
        # and all prices
        conversion_of_current_best = [i[j] for i,j
                    in zip(self.means, self.currentBestArms)]
        price_of_current_best = np.array([i[j] for i, j
                    in zip(self.prices, self.currentBestArms)])
        num_product_sold_of_current_best = np.array([i[j] for i, j
                    in zip(self.num_product_sold_estimation, self.currentBestArms)])
        nearbyRewardsTable = np.zeros(self.prices.shape)
        nodesToVisit = [i for i in range(len(self.prices))]
        for node in nodesToVisit:
            for price in range(len(self.prices[0])):
                nearbyRewardsTable[node][price] =
                    sum(self.visit_probability_estimation[node][price]
                        * conversion_of_current_best * price_of_current_best
                        * num_product_sold_of_current_best
                        * self.means[node][price])
        return nearbyRewardsTable
```

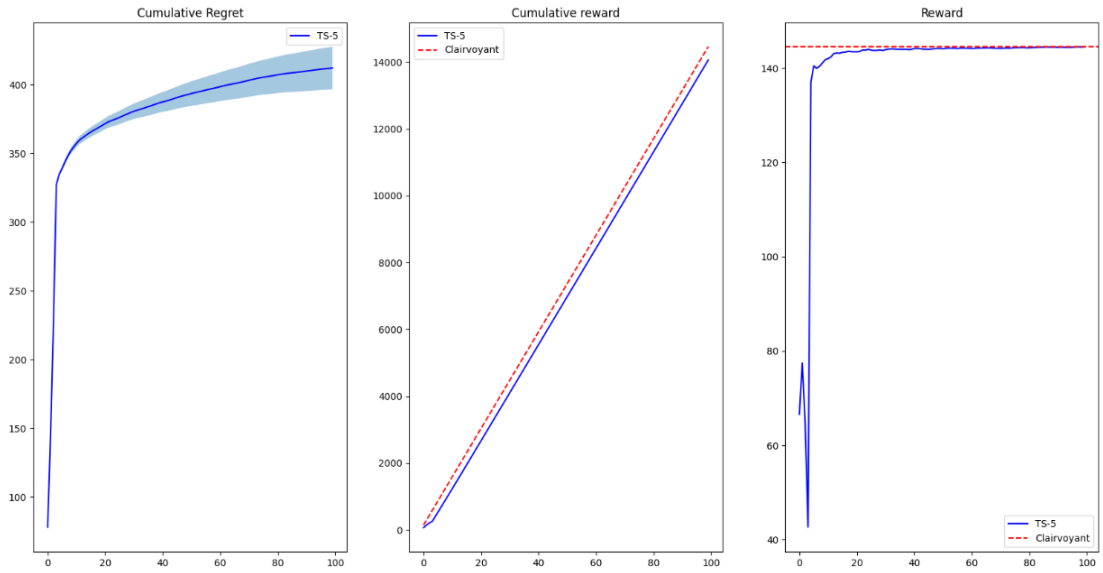
5.1.1 Results

Although the scenario provides less knowledge to the learner, the latter converges with almost the same performances of step 4, having sufficiently good estimation to have a low cumulative regret.

An interesting point to observe is that the estimations in this scenario are incredibly consistent, having the expected reward that is often very close to the clairvoyant one. For example in UCB-1 with the same study parameters of the previous algorithms, the reward is often an horizontal line with the same value of the clairvoyant, and it is due to the fact that the nearby reward estimation is much more precise than using a simple Monte Carlo Simulation, and it has practically no oscillations in this scenario. However there are some peaks but they appear simply because this method tests again arms with a bad mean value over time, so once a suboptimal super arm is chosen, it will have a worse performance than the optimal, leading to the downside peaks. Overall this criterion seems very accurate also in this scenario, and could be considered even better since it is way faster time wisely.



Similar results are also obtained with the TS method, without the down peaks that appear in the UCB since the TS method doesn't try worse arms over time, leading to an improvement on the cumulative reward.



6 Optimization with non stationary demand curve

In this scenario, the demand curves could be subjected to abrupt changes: it is required to understand the shift moment and to choose the new best arms in a dynamic way. The learner selects the arms and observes the reward coming from the environment and a new actor is introduced: the change detector that takes in input the reward too and communicate the warning about the shift of the demand curve to the learner.

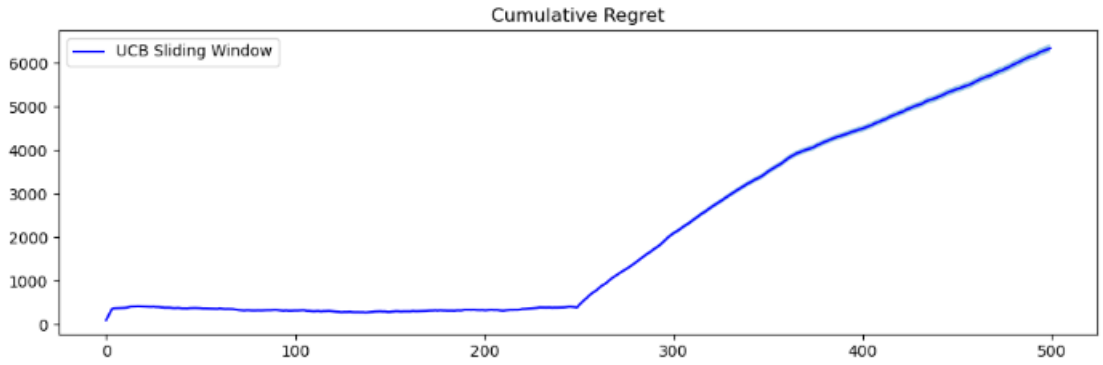
6.1 UCB - Sliding Window

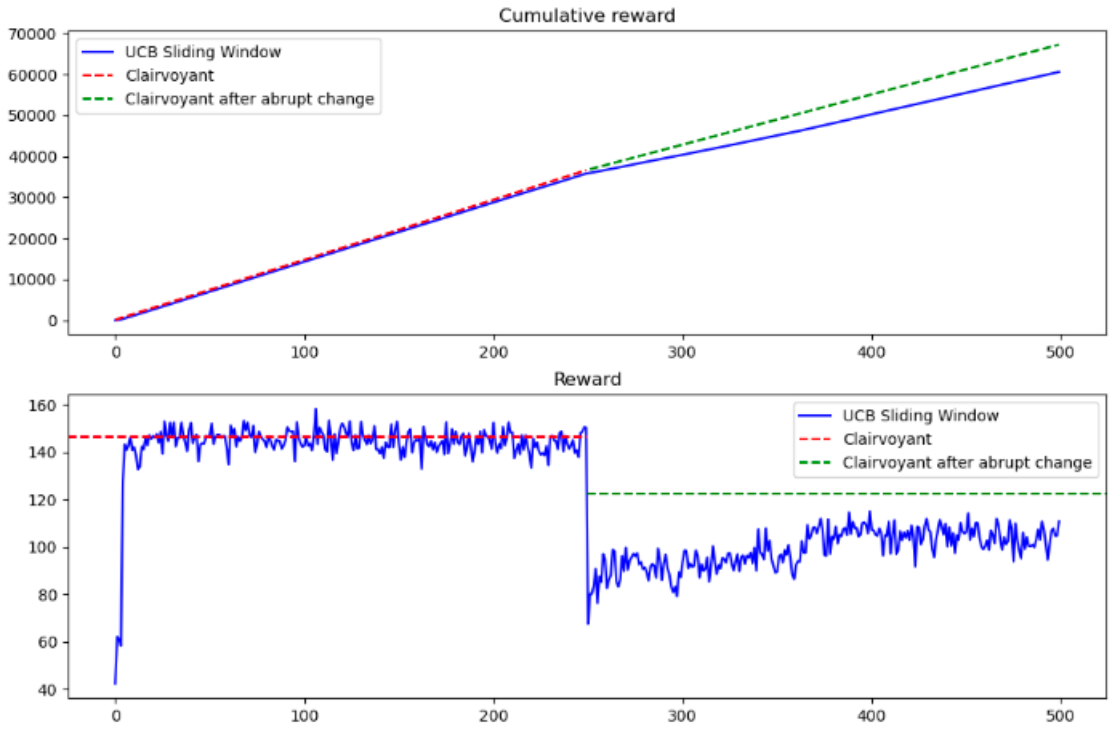
The UCB algorithm with sliding window works in the same way as the UCB explained previously, with the difference on how to compute the means and widths. They are computed using just the samples obtained in the τ previous days. So, basically, using a sliding window approach all the old samples are forgotten, and this is a way of managing abrupt changes. The size of the sliding window τ depends only one the time horizon T as follow:

$$\tau \propto \sqrt{T}$$

6.1.1 Results

In the graphs below a time horizon of 500 days is used and the abrupt change is set at day 250 as clearly visible from the graphs. The size of sliding window τ is 22 days which is the square root of the time horizon. After the abrupt change the reward drastically decrease, but after some days it gets closer to the new clairvoyant line which is different from the clairvoyant before the abrupt change. After the abrupt change it takes some time to discard the old sample, depending on the length of the window size.





6.2 UCB - Change Detection

The UCB - Change detection algorithm can detect a shift in the demand curve taking into consideration the reward of each single arm per product, if the distribution of that rewards suddenly deviates from its average value, an abrupt change could have happened. To accomplish that task, each arm is associated to a class CUSUM.

6.2.1 CUSUM Class

When a product with a specific price is bought, the CUSUM class is updated for that arm: for the first M times, the function the compute empirical mean over the first M valid samples, this will be considered as the new reference point. For the M+1 successive samples, it checks whether there is a change considering the positive and negative deviation from reference point and the consequential cumulative positive and negative deviation. If such quantities exceed a threshold h, then the change detector notify an abrupt change to the learner.

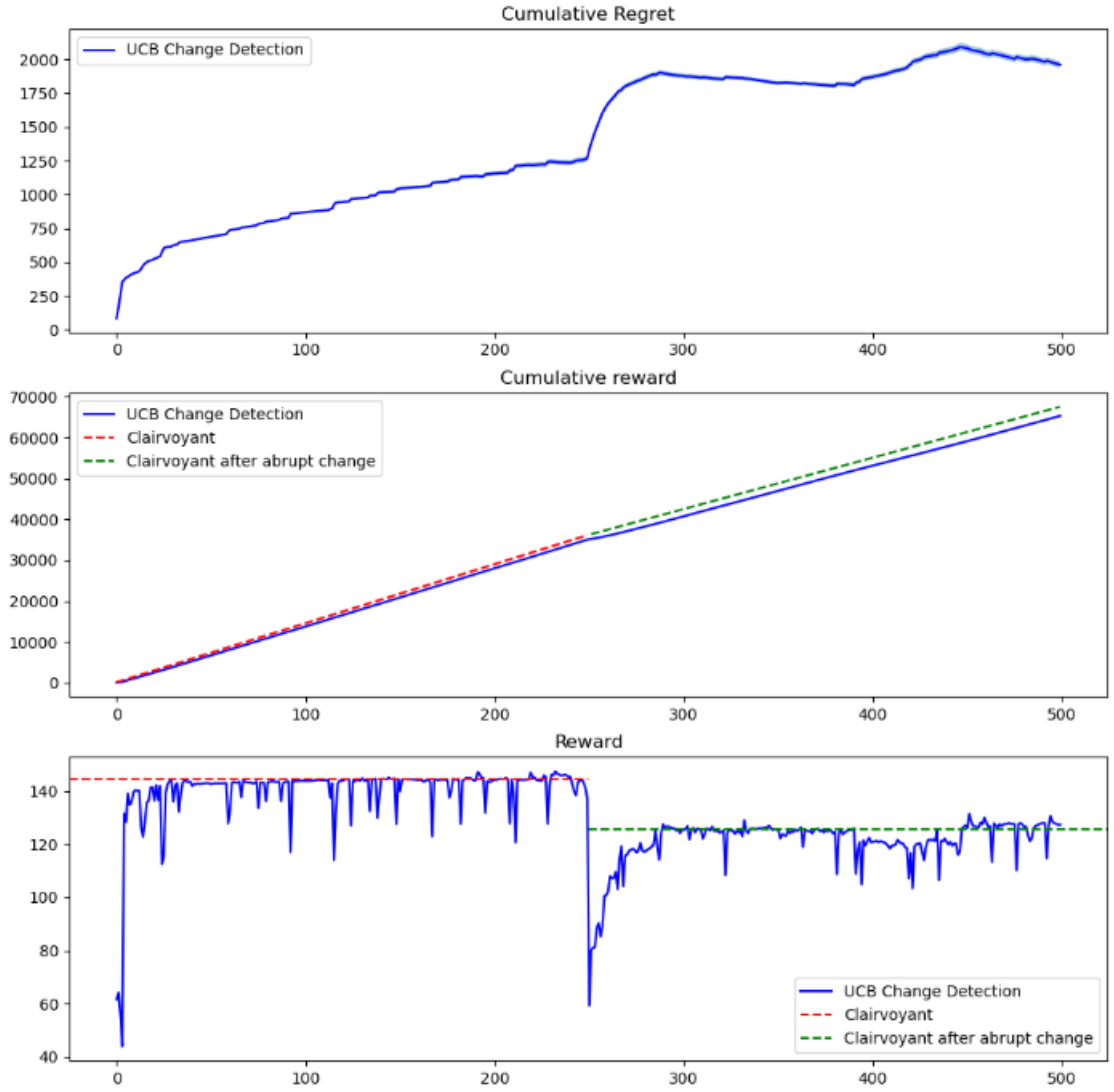
```
    def update(self, sample):
        self.t += 1
        #empirical mean over M, definition of the reference point
        if self.t <= self.M:
            self.reference += sample/self.M
            return False
        else:
            self.reference = (self.reference*(self.t-1) + sample)/self.t
            #positive and negative deviation from reference point
            s_plus = (sample - self.reference) - self.eps
            s_minus = -(sample - self.reference) - self.eps
            #positive and negative cumulative deviation from reference point
            self.g_plus = max(0, self.g_plus + s_plus)
            self.g_minus = max(0, self.g_minus + s_minus)
            #return 1 if the cumulative deviations exceed the threshold
            return self.g_plus > self.h or self.g_minus > self.h
```

6.2.2 UCB after abrupt change and parameter

Once the UCB algorithm has been warned, the valid reward accumulated so far are eliminated, the CUSUM class associated to the arm is reset and the learning process restart. The mean and the widths associated to each arm depends only on the valid reward. M, ϵ and h has been manually tuned.

6.2.3 Results

The curve below shows a regret that suddenly increases due to the abrupt change; after an initial phase of learning, the change detection succeed in pulling the best arms. The simulation here considered 500 number of days, 70 daily interaction and and a shift in the conversion rates in the middle of the considered period.



6.3 Comparison

The change detection algorithm is better then the sliding window one because of the in which the two different algorithms detect the abrupt change. The sliding window requires τ iterations to remove all the old samples, while the change detection algorithm potentially removes them before.
For example in the studied scenario the Sliding Window doesn't converge even 250 days after the abrupt change, while the Change Detection algorithm requires less then 50 days to converge.

7 Context Generation

Until now, we considered all the customers interacting with the e-commerce as part of a unique class, and so the specific behaviors of different classes have not been taken into consideration.

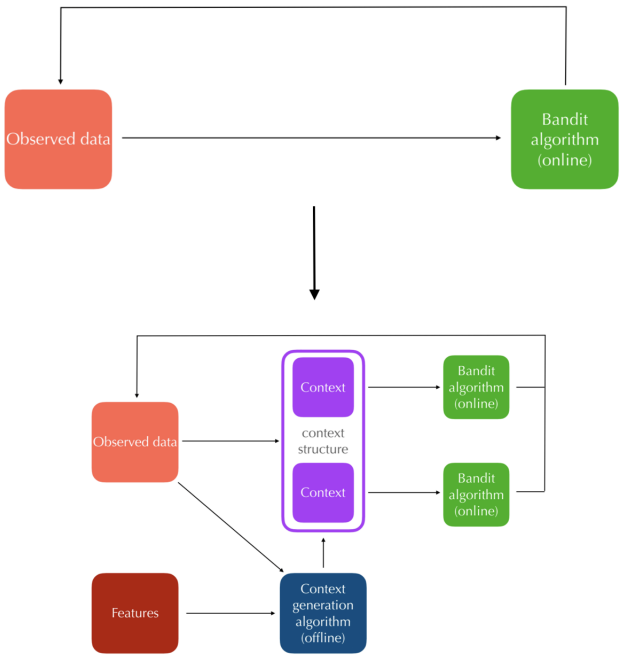
The algorithms of the previous steps do not discriminate between classes, learning through time the aggregated values, but in presence of multiple classes of users the goal is to make the scale, fitting a dedicated model for each type of customers.

7.1 What is a Context?

Given the problem specifications, each user has two binary features that describe her. This allows to obtain at most $2^2 = 4$ different classes as combinations of the features, and 3 of these have been used as classes for the customers:

Class 1: [True, True]
Class 2: [False, False]
Class 3: [False, True]

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



7.2 Context Generation Algorithm

The algorithm used to generate contexts throughout the execution horizon follows the flow reported below:

- For every possible partition of space of the features
  - Evaluate whether partitioning is better than not doing it

This algorithm defines a value for each expandable feature representing the usefulness of the split procedure if applied on this feature, and it is then compared to the non-split case.

The evaluation is performed using the **split condition**:

$$\underline{p_{c_1}} * \underline{\mu_{ac_1{}^*,c_1}} + \underline{p_{c_2}} * \underline{\mu_{ac_2{}^*,c_2}} \geq \underline{\mu_{ac_0{}^*,c_0}}$$

being $\underline{p_{c_x}}$ the probability that context c_x occurs and $\underline{\mu_{ac_x^*,c_x}}$ the lower bound on the expected reward for context c_x.
The value of the lower bound depends on the distribution that, if finite (Bernoulli), can be expressed using the **Hoeffding Bound**:

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

If the condition is satisfied, the contexts c_1 and c_2 are generated splitting c_0.

7.3 Implementation of Context Generation

The implementation of the algorithm strictly follows the theoretical description.

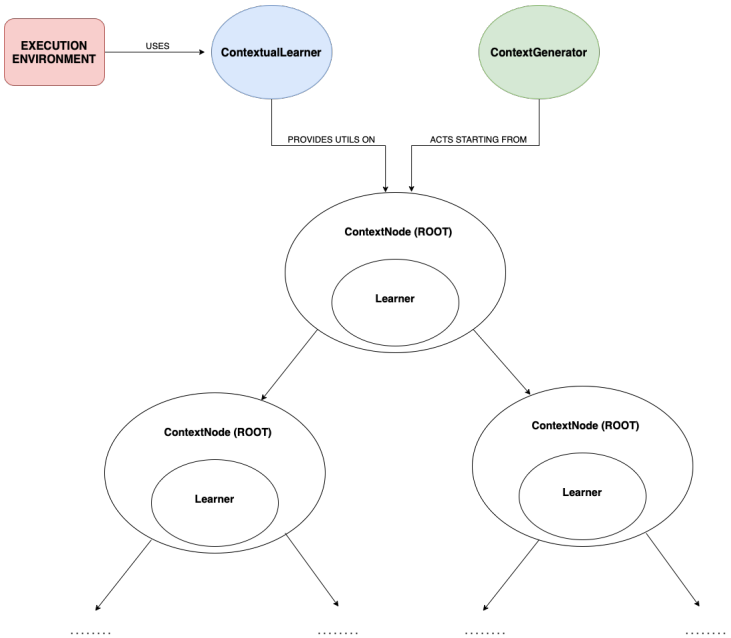
At the beginning, the program starts working with a single learner treating all the classes.
Every $sd = 14$ days, the split condition is evaluated on all the leaf nodes and, if it is satisfied, two new context nodes are created, linked to the parent node and a new independent learner is attached to each of them.

These new learners do not start from scratch: they are subject to an initial training phase using all the data collected until now, that are strictly related to the new node's features constraints.

The procedure of split check is repeated until the end of the execution or whether the features have been fully described in terms of contexts (no more leaves have expandable features).

Technically speaking, the different contexts are organized in a *ContextNode*'s tree, in which each node contains its specific learner, a *ContextGenerator* instance is the one providing all the methods needed to evaluate and successfully perform the split and a *ContextualLearner* class is used during the execution to make use of utility functions on the data structure provided by it.

A visual representation is reported here:



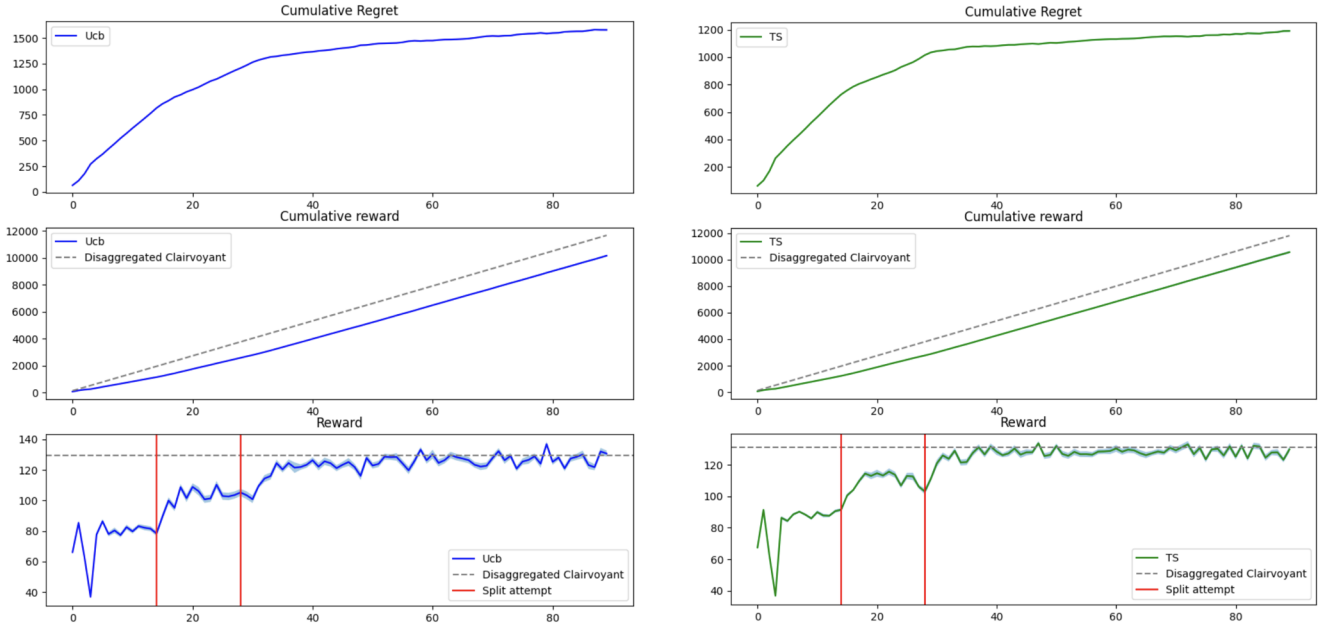
7.4 Results

Differently from the aggregated case, the performance of the program while working with different classes of users must be studied considering different pursuit curves, provided by the *disaggregated Clairvoyant* model.

The quality of the algorithm is evaluated by running *Context-UCB1* and *Context-TS* for a time horizon of T=90 days and plotting three different metrics:

1. Cumulative regret
2. Cumulative reward
3. Average daily reward

Their graphical representation is reported below:



The difference compared to the single class cases is the substantial enhancement of the average daily reward obtained following the contribution of the context generation algorithm, respectively on days 14 and 28. It is fair to specify that, in this case, the split check is immediately successful at the first two attempts, as each day consists of a number of user interactions, belonging to potentially different contexts, which never drops below 100. This method doesn't always split so fast and sometimes doesn't even split based on the samples received and the amount of them. So it's necessary to have a fair amount of user interactions to guarantee a proper splitting.