ONLINE LEARNING APPLICATIONS PROJECT

A. Y. 2021/2022

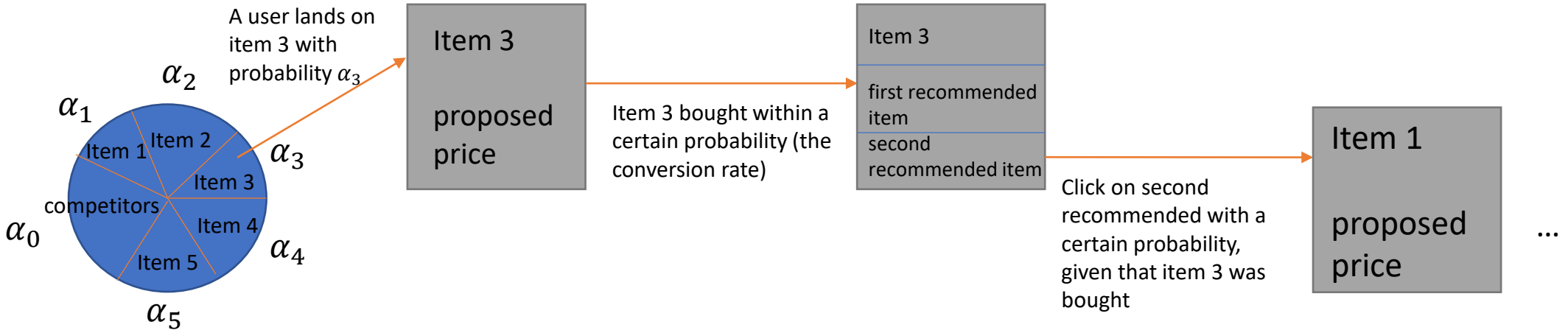
Team Members:

- Antonio Ercolani
- Francesco Romanò
- Andrea Lentini
- Ledio Sheshori

Topics chosen: Social influence + Pricing

The problem

The task is a pricing problem that exploits also social influence techniques. We must select the price for 5 different items while considering that each item bought influence the probability to buy two other items that are recommended after the purchase. So to properly model this situation we had to model a reward that keeps track of all the items bought by a user in a single session, and we must also choose prices considering this influence.



Optimization algorithm

The objective function is the maximization of the **cumulative expected margin**. **Greedy algorithm** that optimizes the objective function when all the parameters are known.

```
while True:
    cumulative_expected_margins = []

    for idx, element in enumerate(optimal):

        if element == self.n_prices - 1:
            cumulative_expected_margins.append(0)
            continue


        candidate = np.array(optimal)
        candidate[idx] = element + 1


        margin_increment = max(0, self.compute_margin(candidate) - optimal_cumulative_expected_margin)
        cumulative_expected_margins.append(margin_increment)


    candidate_product = np.argmax(cumulative_expected_margins)


    if cumulative_expected_margins[candidate_product] == 0:
        break

    optimal[candidate_product] += 1
    optimal_cumulative_expected_margin += cumulative_expected_margins[candidate_product]
```

If the assignment is not feasible, it leads to zero increment, so continue with other possible assignments.

The assignment is feasible, so assign the price to the product.

Evaluate the marginal increment of this assignment.

Chose the candidate providing the best marginal increase. If no new configuration is better than the previous one, then stop.

Optimization algorithm

As expected, this greedy algorithm:

- Monotonically increases the **prices** and the **cumulative expected margin**.
- Doesn’t cycle.
- Gives no guarantee that the given solution is the optimal one.

```
prices = np.array([[1, 3, 5, 7],
                   [1, 3, 5, 7],
                   [1, 3, 5, 7],
                   [1, 3, 5, 7],
                   [1, 3, 5, 7]])

conversion_rates = np.array([[0.2, 0.3, 0.1, 0.4],
                             [0.1, 0.4, 0.2, 0.3],
                             [0.6, 0.1, 0.1, 0.2],
                             [0.1, 0.1, 0.1, 0.7],
                             [0.2, 0.2, 0.5, 0.1]])

n_items_to_buy_distr = np.array([[10, 2],
                                 [10, 2],
                                 [10, 2],
                                 [10, 2],
                                 [10, 2]])

learner = Greedy_Learner(prices, conversion_rates, n_items_to_buy_distr)
print(learner.pull_prices_activations())
```

```
(base) PS C:\ola-pricing> & C:/Users/andre/anaconda3/python.exe c:/ola-pricing/project/src/step3/greedy_test.py
[1 1 0 3 2]
(base) PS C:\ola-pricing>
```

Probability matrix

The probability matrix, that determines the behavior of a user belonging to a certain class in the website, is sampled randomly satisfying the primary-secondaries mapping decided by the business unit.

```
def prob_matrix_generation(primary_to_secondary_mapping, lambda_parameter):
    prob_matrix = np.zeros((3,5,5))
    for user_class in range(0,3):
        for item in range(0,5):

            first_secondary = primary_to_secondary_mapping[item][0]
            to_first_secondary_prob = np.random.uniform(0, 1)
            prob_matrix[user_class][item][first_secondary] = to_first_secondary_prob

            second_secondary = primary_to_secondary_mapping[item][1]
            to_second_secondary_prob = np.random.uniform(0, 1) * lambda_parameter
            prob_matrix[user_class][item][second_secondary] = to_second_secondary_prob

    return prob_matrix
```

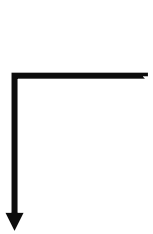
For every primary we sample the probability of clicking on the corresponding secondary from a **uniform distribution**.

The **second secondary** probability is multiplied by the **lambda parameter**

Alpha ratios

The alpha ratios, that are responsible for the starting page of the users, are sampled every day from independent Dirichlet random variables

```
alpha_parameters = [[2,2,3,4,5,6],
                    [2,2,3,4,5,6],
                    [2,2,3,4,5,6]]
```

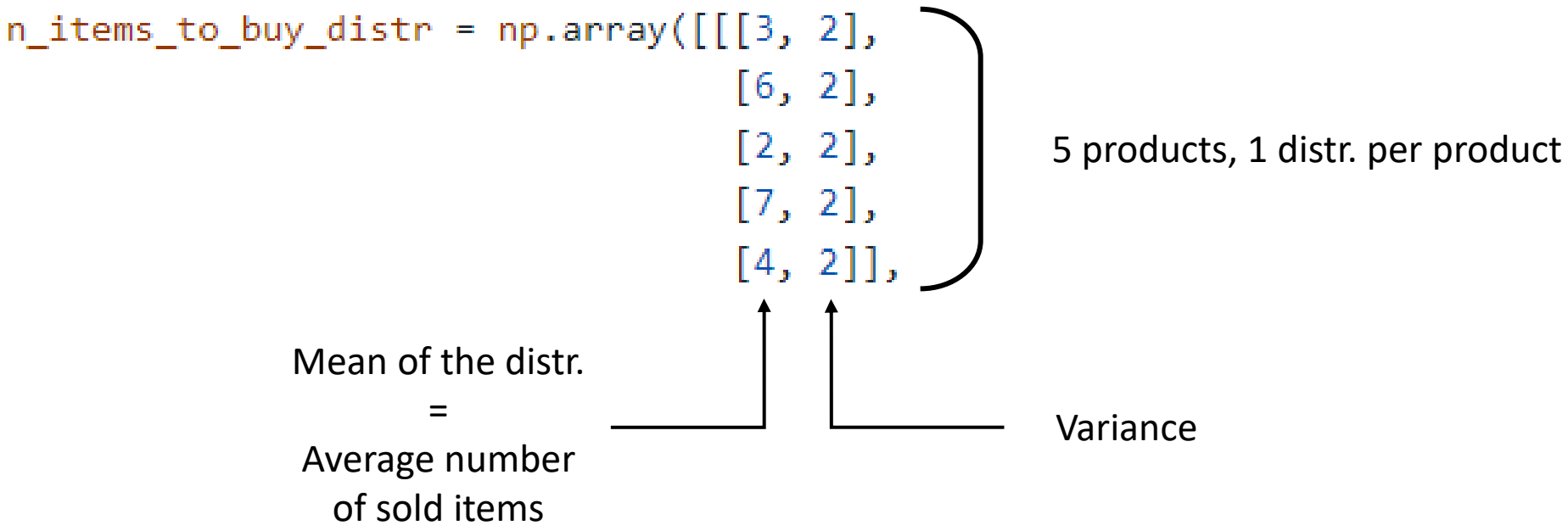


```
def alpha_generation(alpha_parameters, seed):
    alphas = np.zeros((3,6))
    for user_class in range(0,3):
        alphas[user_class] = dirichlet.rvs(alpha=alpha_parameters[user_class], size=1, random_state=seed)
    return alphas
```

Number of sold items

The number of sold items per product is modelled as a gaussian distribution.

In the environment, when a user decides to buy a product we sample the number of sold units from the associate gaussian distribution.



Simulator

The following pseudocode shows an high level overview of the steps made by the simulator

```
foreach simulations:
    foreach day:
        today_prices = bandit.pull_prices()
        foreach user:
            user_class = Retrieve user class from Feature_1 and Feature_2 realization
            starting_point = Sample the starting point from the ALPHAS
            items_to_visit = []
            items_to_visit.append(starting_point)
            foreach item in items_to_visit:
                compute whether the user buys the item sampling from the corresponding conversion rate
                compute the number of purchased units sampling from the n_items_to_buy_distr (prev. slide)
                visited_secondary = sample from the prob matrix
                items_to_visit.append(visited_secondary)
```

User reward and regret

For each user we collect all the reward that he generates during the purchases on the website and we compute the regret by comparing it with the estimated optimum reward that a user of that class, starting from a certain item, could have generated

for each user:

user_class = Retrieve user class from Feature_1 and Feature_2 realization

starting_point = Sample the starting point from the ALPHAS

user_reward = Collect the reward generated by the all the purchases of the user

user_opt = opt_per_starting_point[user_class][starting_point]

user_regret = user_opt – user_reward

Monte Carlo estimation

In all the project we exploit the influence between items. This information is encoded in the activation probabilities:

$A_{x->y}$: Activation probability from item X to item Y

We need to estimate them, Monte Carlo estimation algorithm allows to do this through multiple exploration simulations:

- For each item, consider it as a seed and simulate a random walk according to the given probability graph of items influence
- After collecting a sufficient number of simulations, estimate activation probabilities from the considered seed *i* to an item *j* as the ratio:

#times j is visited from seed i / #simulations

Monte Carlo estimation

At the end of this algorithm we obtain a matrix with a row for each item (considered as a seed) and a column for each item (considered as destination from the seed. Each cell of this matrix contains the estimated activation probability *i.e.* the probability of reaching an item starting from a specific seed.

Theoretical guarantees over estimation goodness presented during the course were used to select the number of Monte Carlo iterations:

$$R = O\left(\frac{1}{\varepsilon^2}\log(|S|)\log\left(\frac{1}{\delta}\right)\right)$$

- With probability of at least 1-δ the error over the estimate is $\pm\epsilon n$
- S set of seeds, n number of nodes

We have set 1-δ to 95% and ϵn to 5%

User reward and regret – OPT

opt_per_starting_point is a 5x3 matrix in which we compute the **optimal reward that could be generated by a user** belonging to a certain class and starting from a certain item during a visit on the website.

The estimated optimal rewards are computed as follows:

Item 1: Starting item
P_x: Price item of X
C_x: Conversion rate of item X
N_x: Average sold units of item X
$A_{x->y}$: Activation probability from item X to item Y
(Probability of reaching item Y starting from item X)

foreach starting_point:
    foreach user_class:
        rewards = []
        foreach combination of arms: #4^5 = 1024 possible combinations
            rewards.append($P_1*C_1*N_1 + A_{1->2} *P_2*C_2*N_2 + A_{1->3} P_3*C_3*N_3 + A_{1->4} *P_4*C_4*N_4 + A_{1->5} *P_5*C_5*N_5$)

        opt_per_starting_point[starting_point][user_class] = max(rewards)

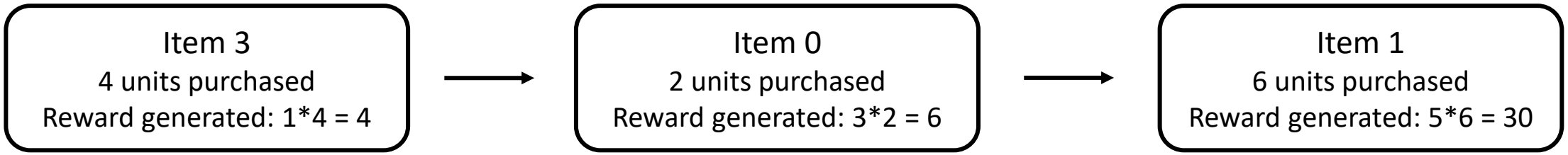
These values are computed only one time before the beginning of the simulations (whether the parameters don't change) using the ACTUAL parameters

User reward and regret - Example

Here we report a numerical example, suppose that:

- The user belongs to class 2
- The starting item is item 3
- The prices pulled by the bandit for the example are [3, 5, 2, 1, 6] (item 0, item 1, … , item 4)

- The user does these 3 purchases and then he stops the visit on the website



Reward = 4 + 6 + 30 = 40

opt_per_starting_point[3][2] == 55

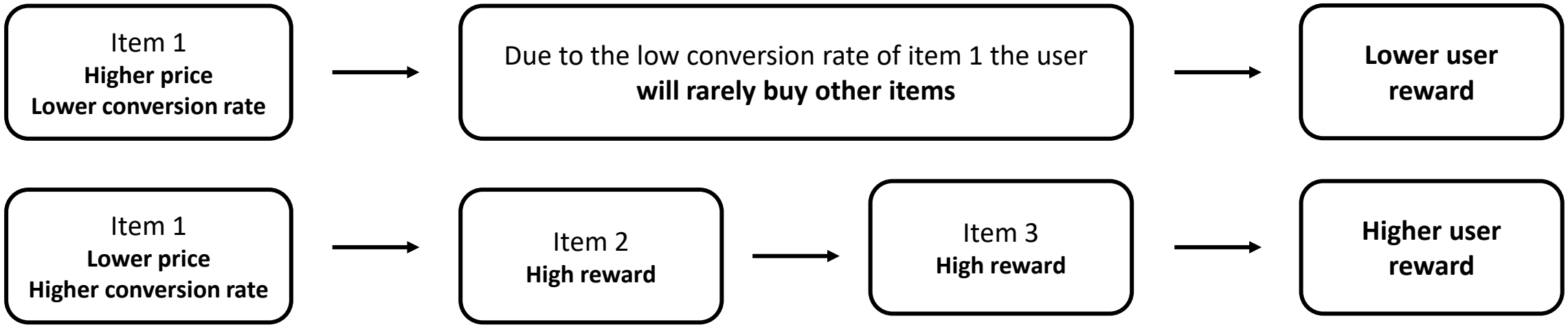
Regret = 55 - 40 = 15

Estimated optimal reward for a user of class 2 starting from item 3

Bandit - Pull arm

Since the behavior of the user in the graph (the purchase of a series of products) is similar to that of the social influence we built a pull arm strategy that **for each item takes into account the reward provided by the item that we are considering and the subsequent rewards provided by the items that the user could buy afterwards.**

For example there could be a situation like the following:



Could be better to lower the price on certain items increasing their conversion rate since the subsequents items could bring high rewards

Bandit - Pull arm

The algorithm works by estimating the rewards generated by a user during a visit on the website for every combination of the items' arms. Then **the combination or arms that provides the best estimated reward is the one pulled by the bandit**.

The reward estimation is done in this way (similar to the OPTIMUM computation):

$$\begin{aligned} reward(arms\_combination) = {} & P_1 * C_1 * N_1 + \\ & A_{1->2} * P_2 * C_2 * N_2 + \\ & A_{1->3} * P_3 * C_3 * N_3 + \\ & A_{1->4} * P_4 * C_4 * N_4 + \\ & A_{1->5} * P_5 * C_5 * N_5 \end{aligned}$$

Item 1: Starting item
P_x: Price item of X - Arm
C_x: Conversion rate of item X - estimated by the bandit
N_x: Average sold units of item X - actual or estimated
$A_{x->y}$: Activation probability from item X to item Y

The subsequents items are weighted by the corresponding activation probability

Bandit - Pull arm

Since the number of rewards to be evaluated is quite a large number (we have to evaluate every possible combination of items' arms) we have developed two similar algorithm to face the problem:

Complete method

The complete method simply evaluates every possible combination of arm per every possible starting point and pulls the arms that generate the higher reward.

Number of combinations: $4^5 * 5 = 5120$

Estimated method

The estimated method **reduces the number of combinations to be evaluated by keeping the arms pulled in the previous day and changing only one arm at a time.** In this way the number of combinations is reduced but **we can change only one arms per day.**

Number of combinations: 5 * 4 * 5= 100

Bandit - Update

As we said in the previous slides **the bandits estimates the conversion rate** of the products and not directly the rewards. Therefore the ‘rewards’ that we pass to them are only the outcomes of the purchases (items bought or not).

Obviously the nature of the bandit will influence the estimation process as described below.

UCB

In the UCB bandit we estimate the conversion rates using means and bounds.

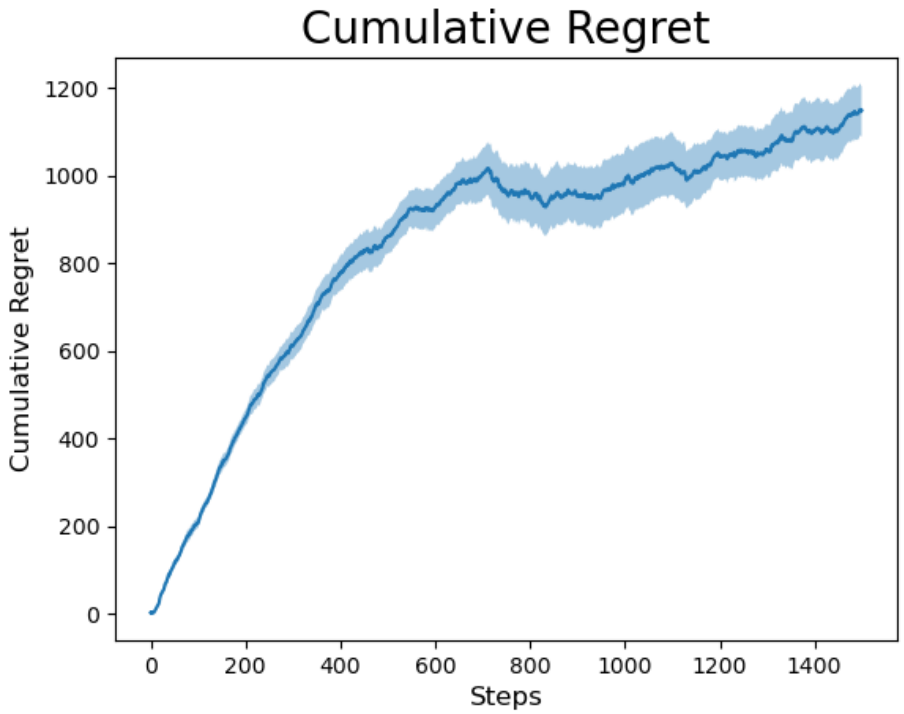
In the maximization formula showed in the previous slides in place of the conversion rates we use mean+bound of the corresponding item-arm.

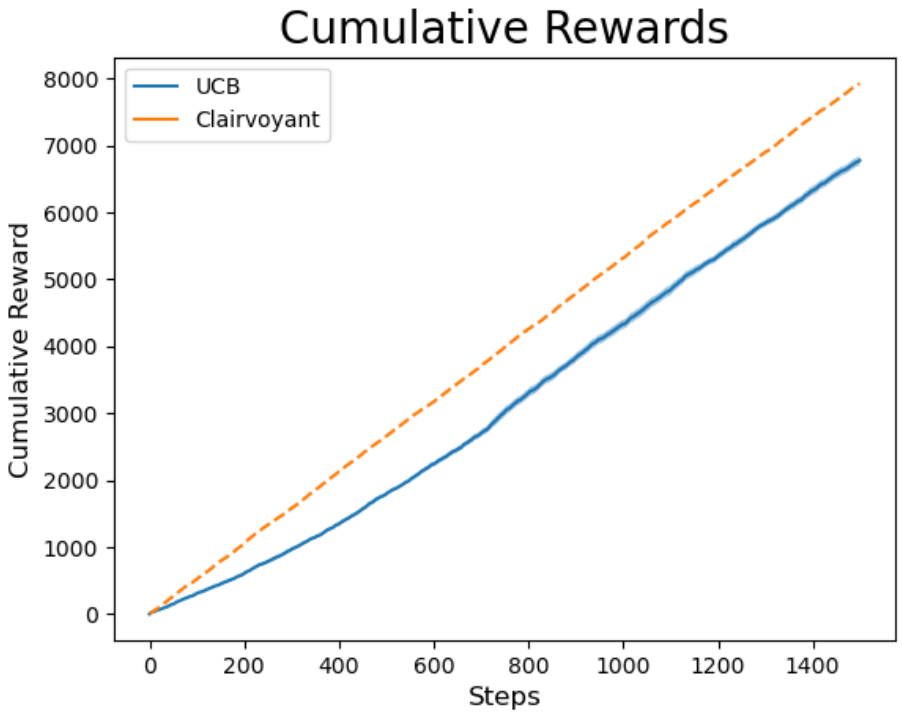
TS

In the Thompson Sampling bandit we estimate the conversion rates by means of the alpha and beta parameters of Beta distributions.

In the formula in place of the conversion rates we simply draw a sample from the Beta using the alpha/beta parameters associated to the item-arm.

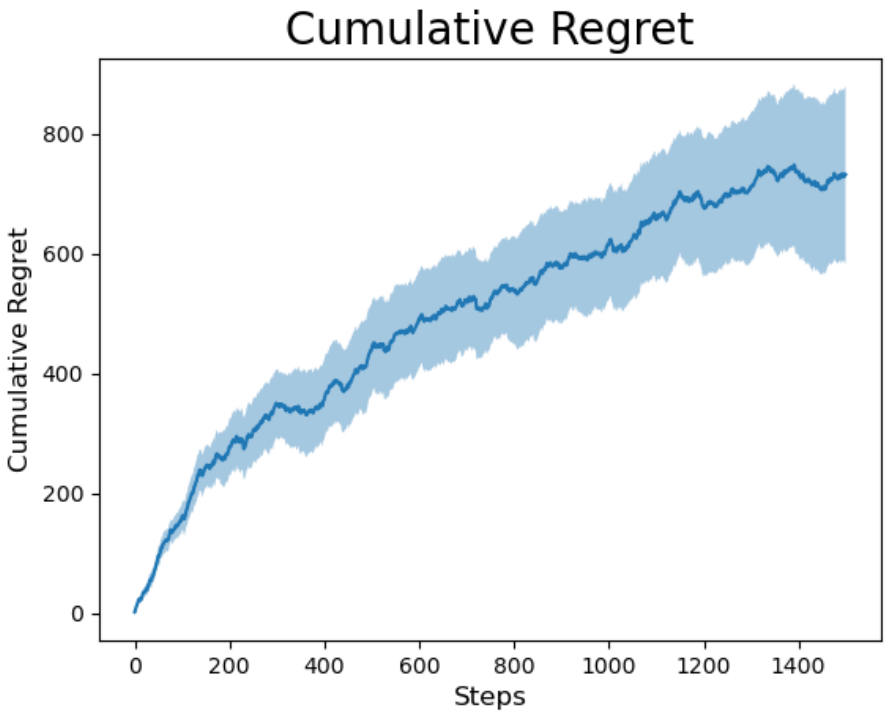
UCB with uncertain conversion rates

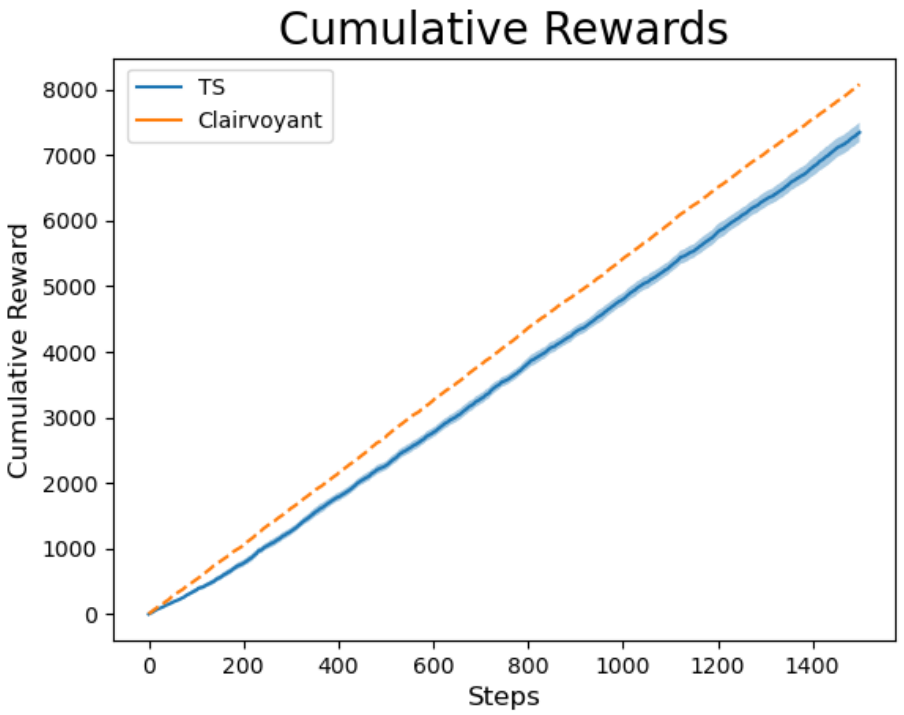




As we can notice the UCB algorithm is able to learn the arms that lower the regret.

TS with uncertain conversion rates





Also in this case the TS algorithm is able to learn the arms that lower the regret. Moreover TS, as we expected from the theory, performs better reaching a lower regret in the number of steps with respect to UCB.

Uncertain α ratios, and number of items sold per product

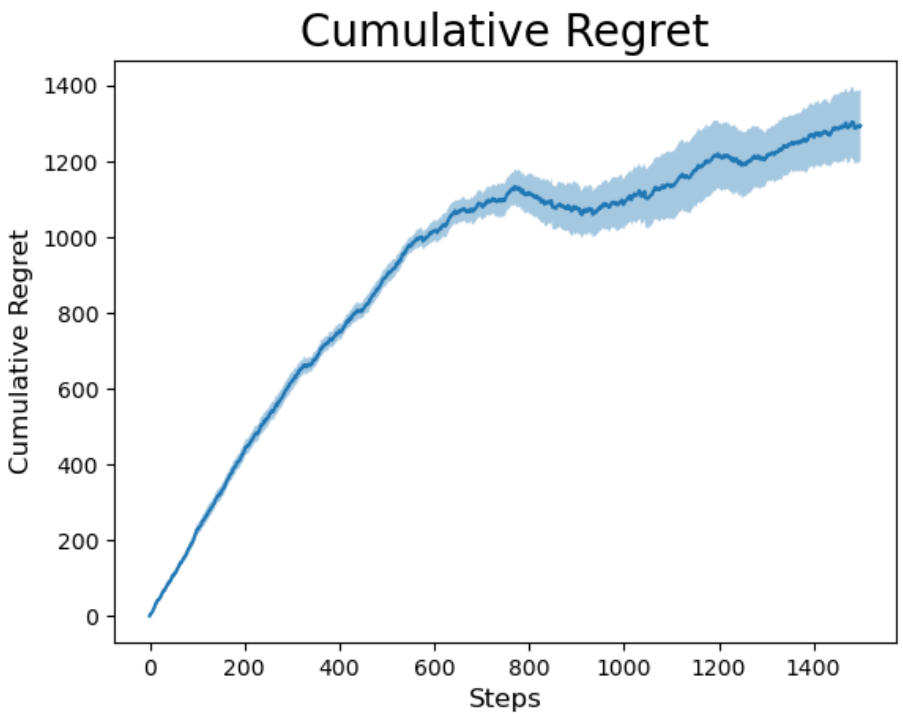
To face this step we did the following:

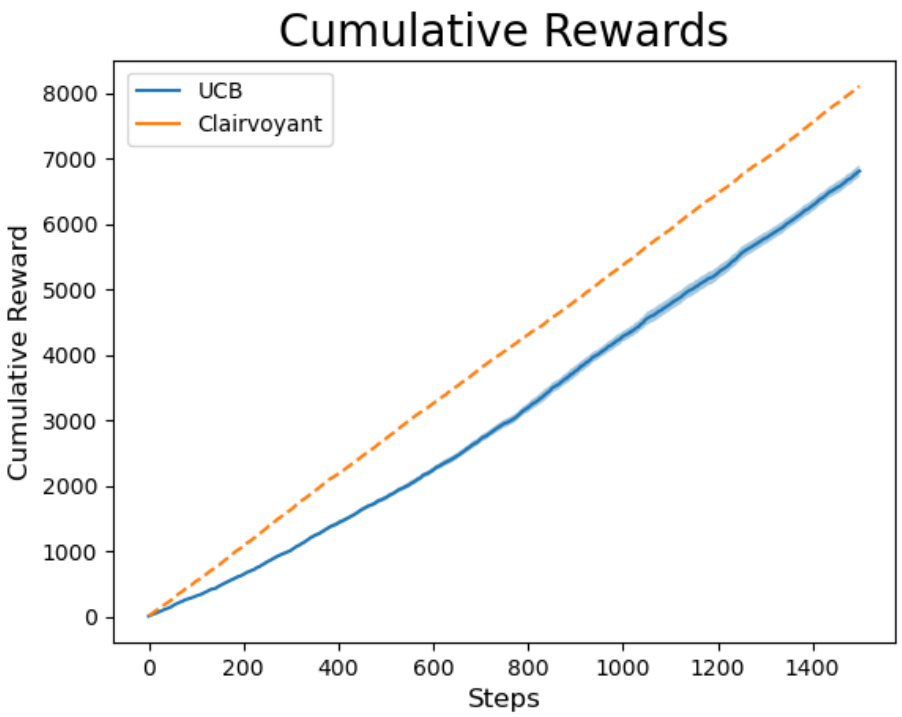
- For what regards the uncertain number of items sold per product we have simply estimated them by collecting the outcomes of the simulator and averaging them.

  For example if for item 1, user class 2, we have observed the following number of items sold: 4, 5, 4, 5, 3, 6. We estimate a mean of (4+5+4+5+3+6)/6 = 4.5 and this is the value that we pass to the bandit for the pull arm maximization.

- Instead for the alpha ratios, since only the simulator (that knows the actual parameters) use them we did nothing.

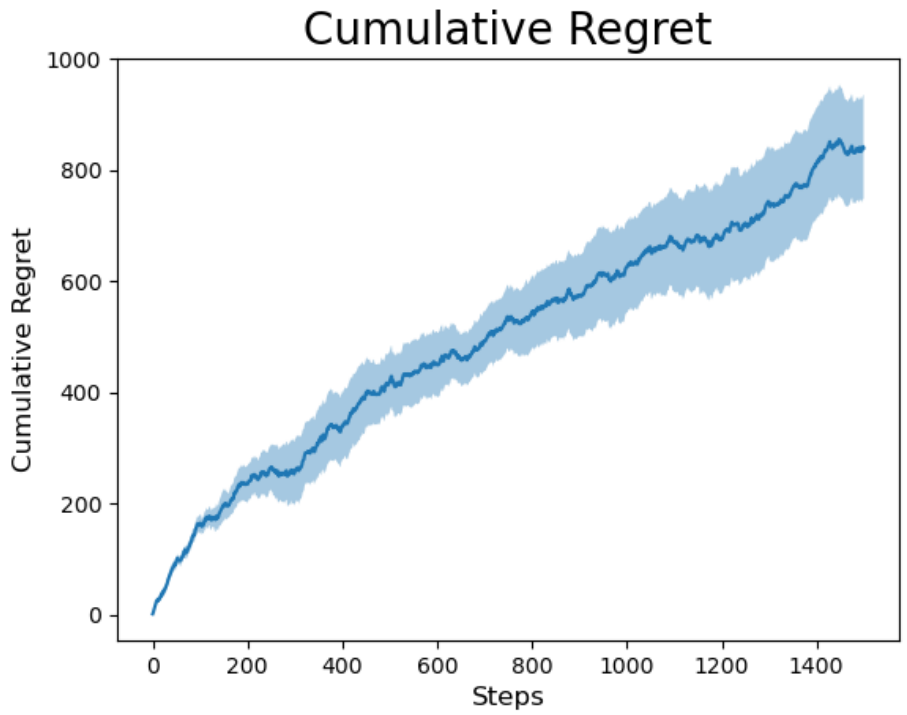
UCB with uncertain number of items sold per product

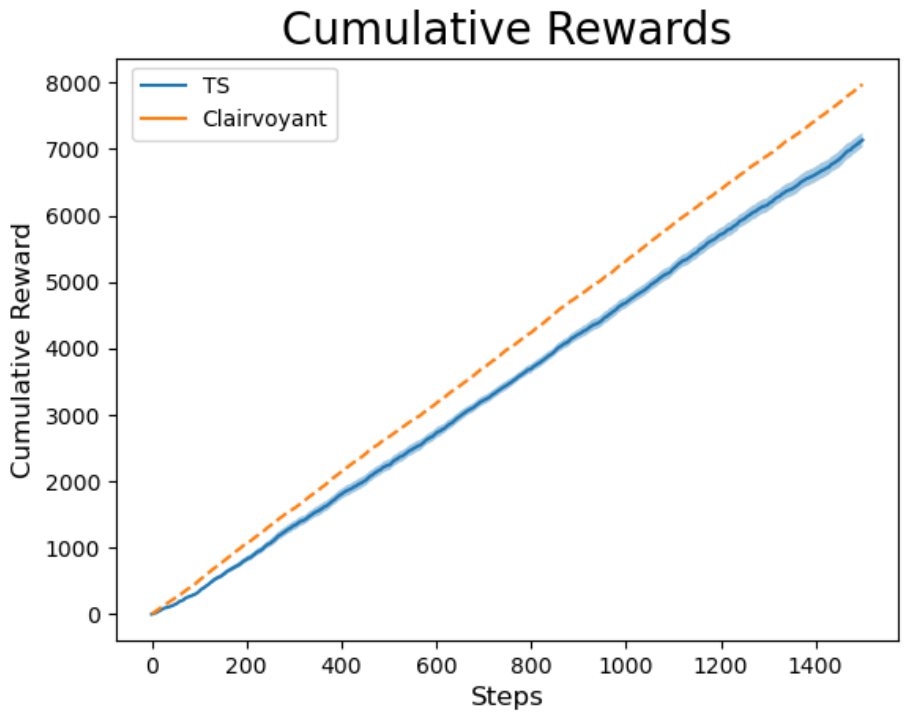




As we can notice the approximation of the number of items sold per product, that is used in the pull arm maximization, brings a drop in the performance (higher regret compared to the standard UCB)

TS with uncertain number of items sold per product





The same holds for the TS case

Probability matrix estimation

To estimate the probability matrix when the graph weights are unknown, we used a technique based on probability matrix inference through the generation of episodes on the graph.

1. **Diffusion**: Collection of simulated episodes on the graph.
2. **Probability estimation**: Estimation of probabilities through credit assignment.

The estimation phase can be done in parallel for each node in the graph, so we implemented a **threaded version** of the second step to speed up the computation.

Probability matrix estimation

1. Diffusion

The nodes in our graph are items, so a diffusion on the graph is a simulation of interactions with the items. We start from a random starting node and then we simulate an **episode**. We use a variable *history* to keep track of the activated nodes in the diffusion phase.

```
while t < n_steps_max and np.sum(newly_active_nodes) > 0:
    p = (prob_matrix.T * active_nodes).T
    activated_edges = p > np.random.rand(p.shape[0], p.shape[1])
    prob_matrix = prob_matrix * ((p!=0) == activated_edges)
    newly_active_nodes = (np.sum(activated_edges, axis=0) > 0) * (1 - active_nodes)
    active_nodes = np.array(active_nodes + newly_active_nodes)
    history = np.concatenate((history, [newly_active_nodes]), axis = 0)
    t += 1
```

Probability matrix estimation

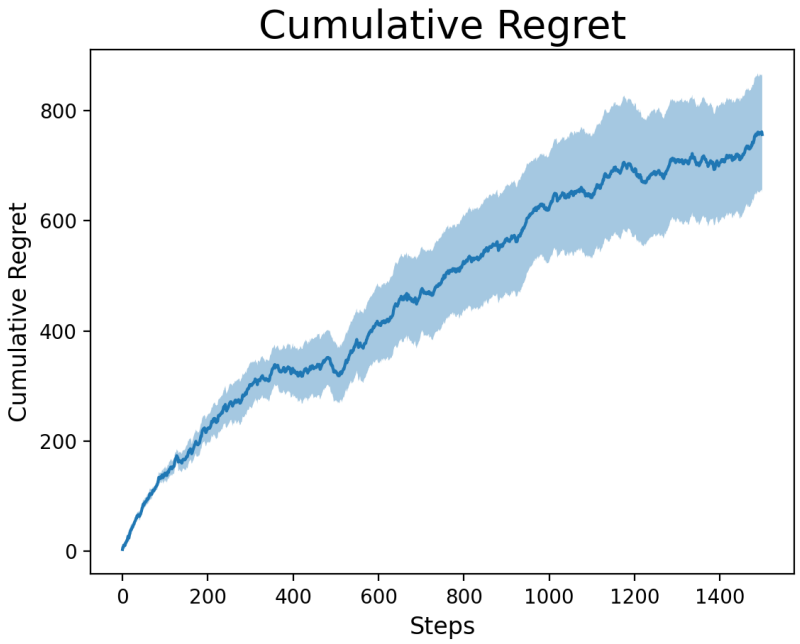
2. Probability estimation

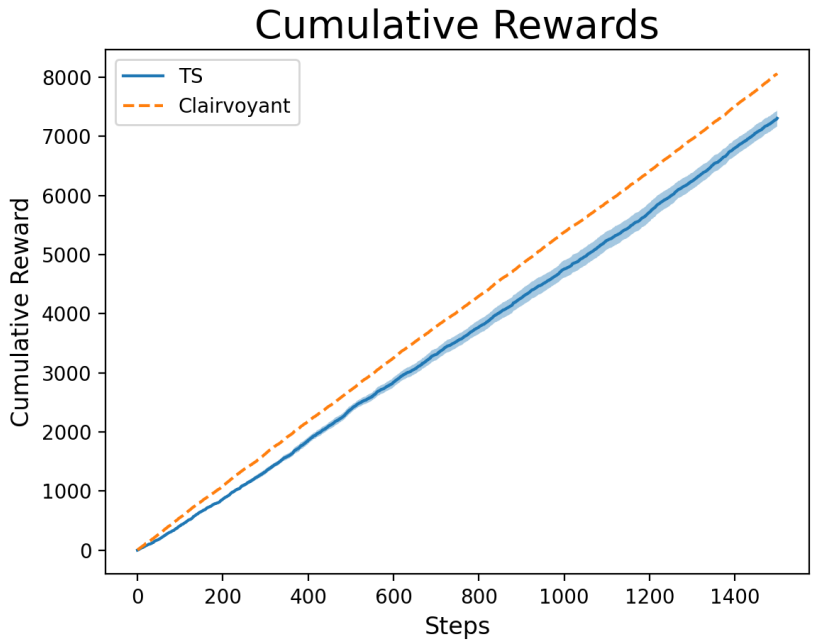
For the final estimation, we exploit a **credit assignment technique**. For each edge (u, v), we compute the final estimation as the division of the sum of the credits assigned to the edge by the number of episodes in which the node u has been active previously than the node v, or the node u has been active, and the node v has not been active.

$$p_{uv} = \frac{\sum credit_{uv}}{A_v}$$

$$credit_{uv} = \frac{1}{\sum_{w \in S} I(t_w = t_v - 1)}$$

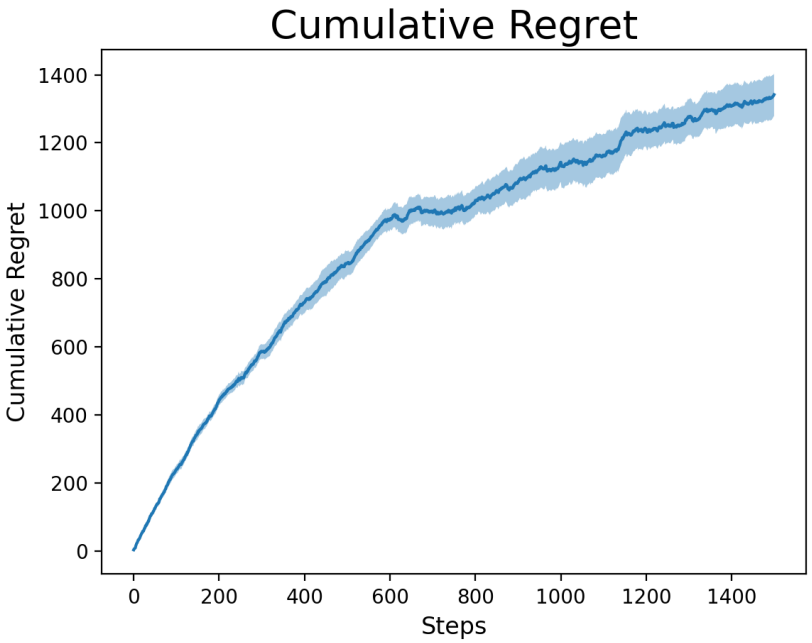
TS with uncertain graph weights

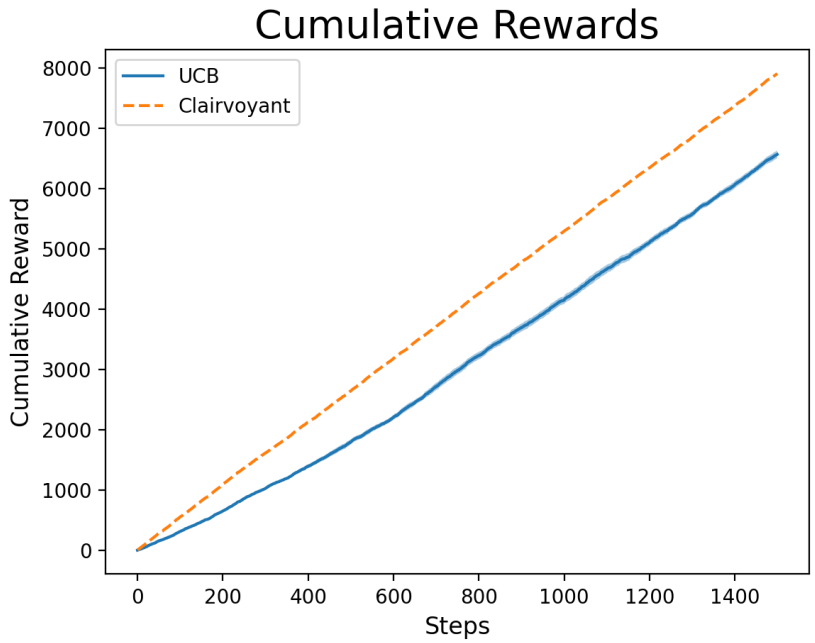




We can see that the algorithm still performs well, the regret is almost the same as the standard case, this means that the probability matrix estimation is done in the right way

UCB with uncertain graph weights





The same holds for the UCB case

Non-stationary environment

Non-stationarity happens when the behavior of the demand curves can change at certain points in time. This is reflected by a change in the values of the conversion rates. To simulate these events, we introduced a *phase* dimension in the conversion rates' matrix - three different **phases**, so modelling two **abrupt changes**.

The standard UCB bandit is not capable of adapting to the changes, so we implemented two bandits that can adjust according to the specific phase, that apply a **sliding-window** and a **change detection** mechanism, respectively.

Non-stationary environment: SW-UCB

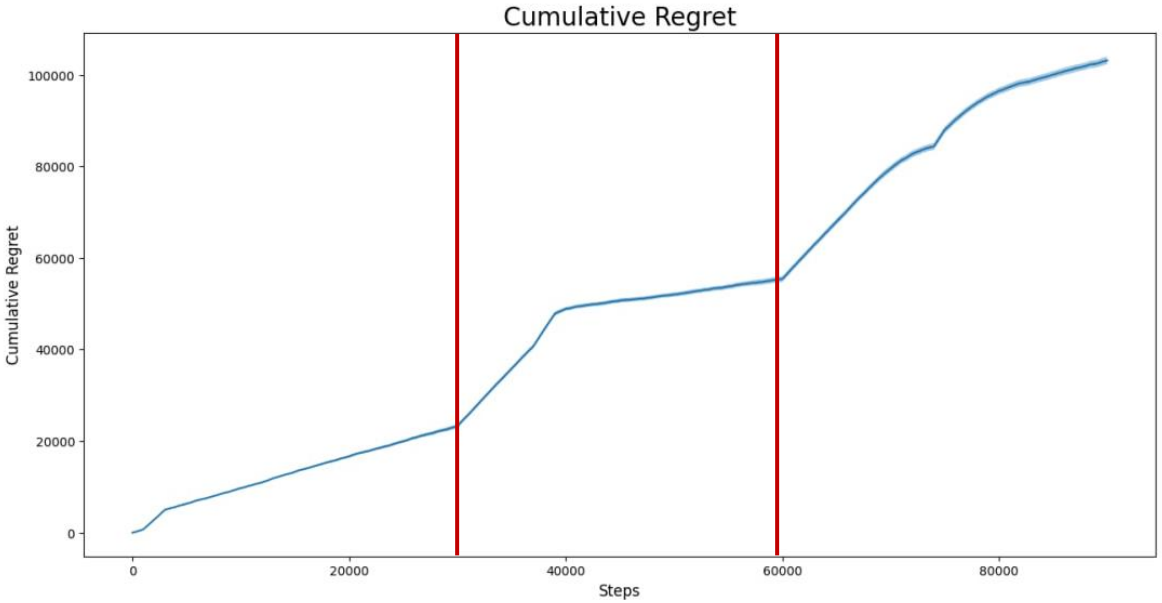
A classic UCB bandit but having a fixed-size memory to keep track only of the last τ collected rewards.

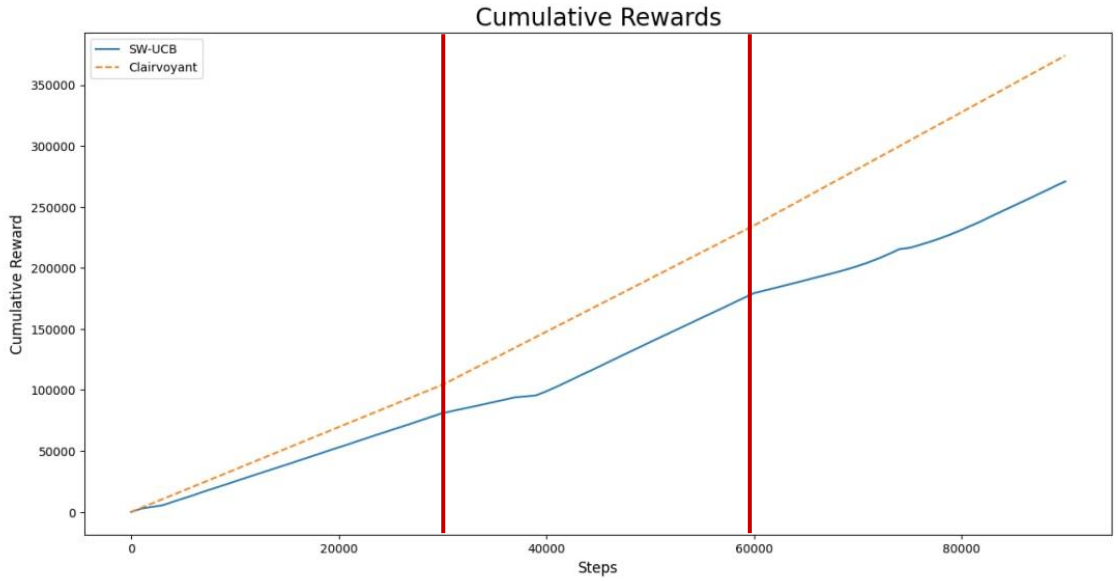
Since we model many users in the system every day, we adapted the standard SW technique to work with multiple rewards (that is, in variable size) collected each day. We used a trick with **placeholders** to support this case.

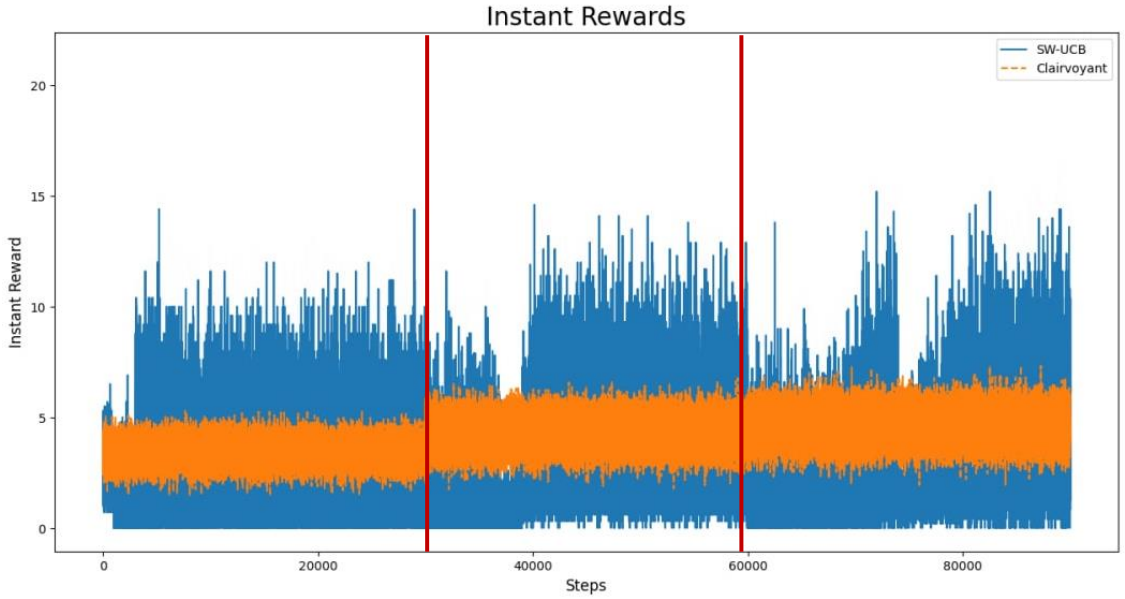
```
for i in range(self.n_items):
    if len(rewards[i]) != 0:
        # nr of rewards collected in the days in window
        nr_rewards_in_window = sum(self.nr_rewards_per_day[i][pulled_arms[i]][-self.window_size:])
        # rewards for item and arm of days in window, excluding placeholders (-1)
        rewards_per_item_arm_in_window = [reward for reward in self.rewards_per_item_arm[i][pulled_arms[i]][-nr_rewards_in_window:] if reward != -1]
        self.means[i][pulled_arms[i]] = np.mean(rewards_per_item_arm_in_window)

for idx in range(self.n_items):
    for idy in range(self.n_prices):
        # widths update with window, same as means
        nr_rewards_in_window = sum(self.nr_rewards_per_day[idx][idy][-self.window_size:])
        rewards_per_item_arm_in_window = [reward for reward in self.rewards_per_item_arm[idx][idy][-nr_rewards_in_window:] if reward != -1]
        n = len(rewards_per_item_arm_in_window)
        if n > 0:
            self.widths[idx][idy] = np.sqrt(2*np.log(self.t)/n)
        else:
            self.widths[idx][idy] = np.inf
```

Sliding Window (SW) UCB with non-stationary demand curve







We highlighted with the red lines the changes in the demand curves.
In the grapsh we can see that the bandit is able to adapt to the changes finding the rights combination of arms after a certain delay bringing a lower regret each time.

Non-stationary environment: CD-UCB

Extension of the UCB bandit with the memory of the collected rewards controlled by a change dection mechanism. If a change is detected for a certain arm for an item, we swipe the collected rewards for that configuration.

```
for i in range(self.n_items):
    if self.change_detection[i][pulled_arms[i]].update(rewards[i]):
        # If a change has been detected for item and arm, empty the rewards and reset the detector
        print(f"Change detected for item {i} and arm {pulled_arms[i]} at time {self.t}")
        self.detections[i][pulled_arms[i]].append(self.t)
        self.valid_rewards_per_item_arm[i][pulled_arms[i]] = []
        self.change_detection[i][pulled_arms[i]].reset()

    self.rewards_per_item_arm[i][pulled_arms[i]] = self.rewards_per_item_arm[i][pulled_arms[i]] + rewards[i]
    self.valid_rewards_per_item_arm[i][pulled_arms[i]] = self.valid_rewards_per_item_arm[i][pulled_arms[i]] + rewards[i]
    self.collected_rewards_per_item[i] = self.collected_rewards_per_item[i] + rewards[i]
    if len(rewards[i]) != 0:
        self.means[i][pulled_arms[i]] = np.mean(self.valid_rewards_per_item_arm[i][pulled_arms[i]])

for idx in range(self.n_items):
    total_valid_samples = sum([len(x) for x in self.valid_rewards_per_item_arm[idx]])
    for idy in range(self.n_prices):
        n = len(self.valid_rewards_per_item_arm[idx][idy])
        if n > 0:
            self.widths[idx][idy] = np.sqrt((2 * np.log(total_valid_samples) / n))
        else:
            self.widths[idx][idy] = np.inf
```

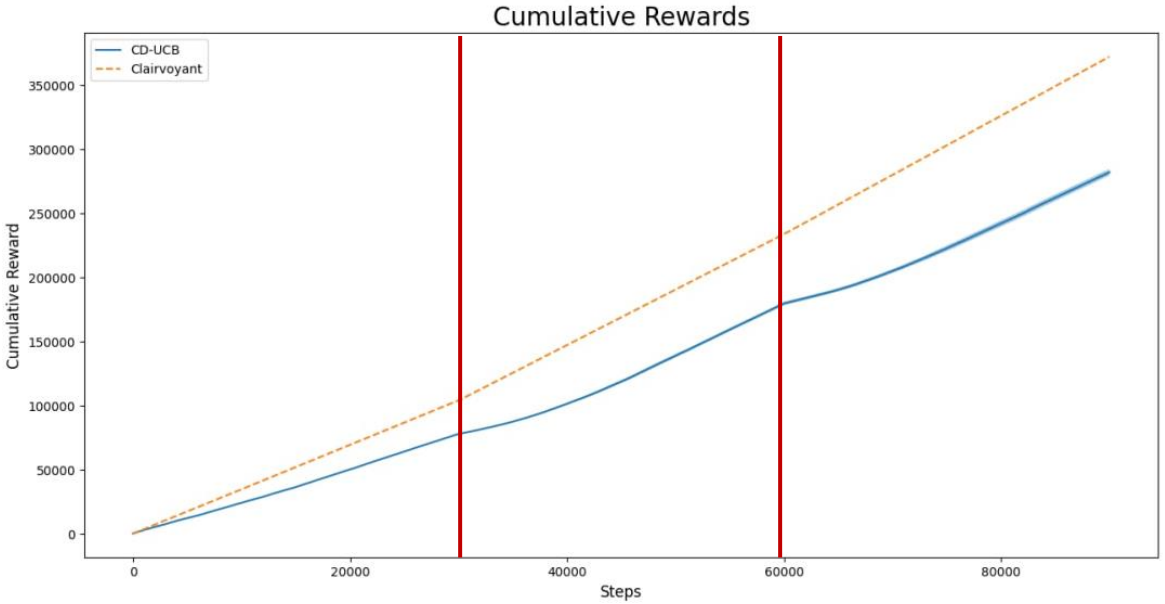
Non-stationary environment: CD-UCB

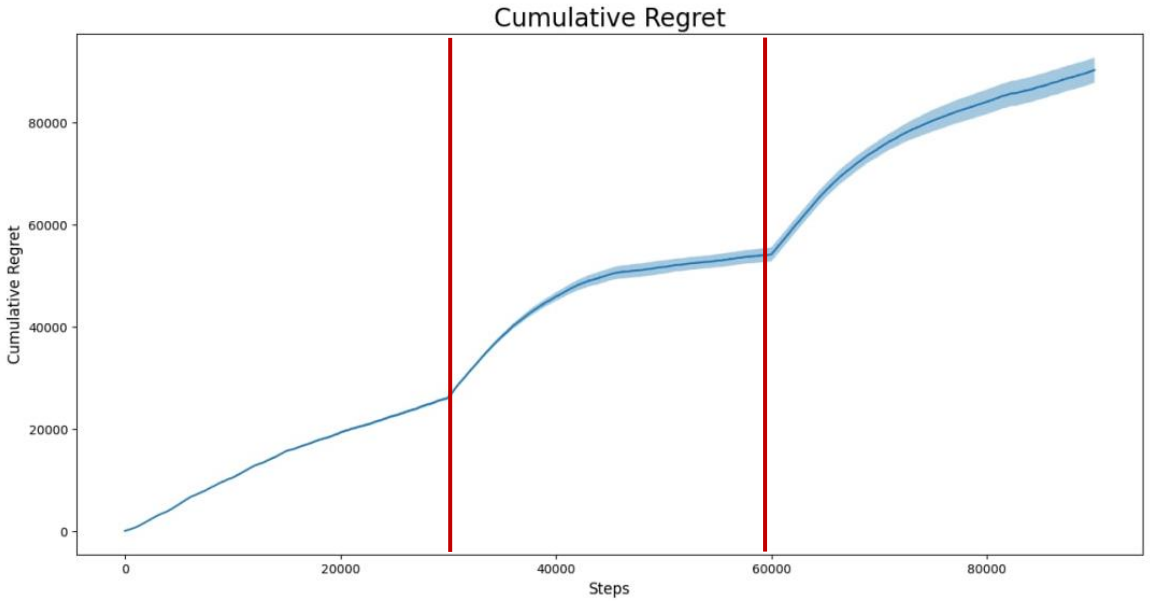
We used the **CUSUM** algorithm, adapted to support multiple rewards in a day through a **majority voting** decision process.

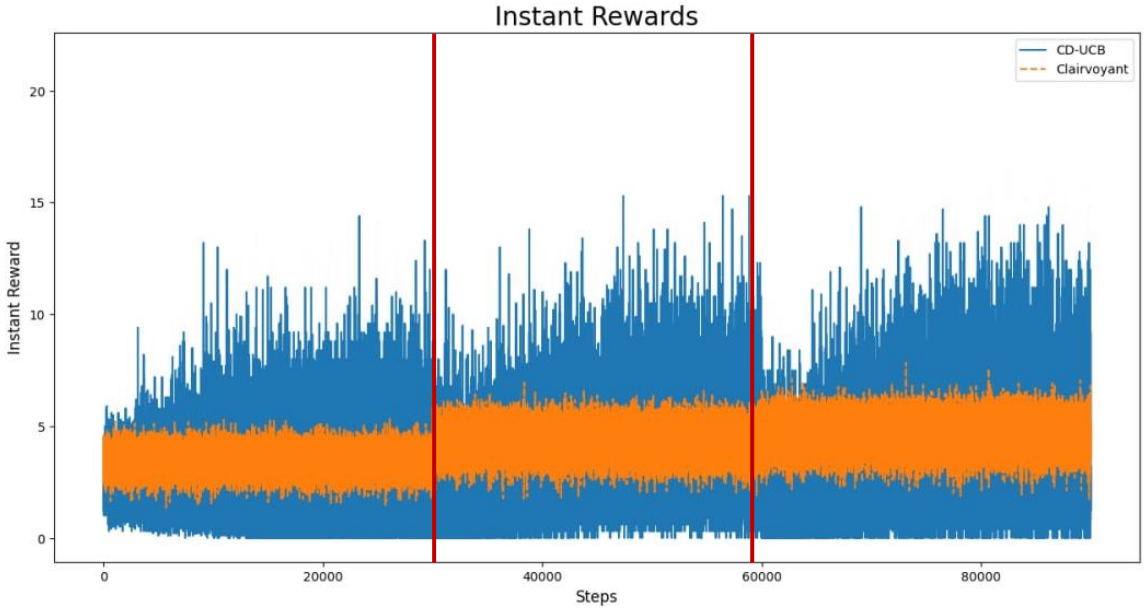
```
# generalization of cusum with multiple rewards
for sample in samples:
    if self.t <= self.M:
        self.reference += sample / self.M
        votes.append(0)
    else:
        s_plus = (sample - self.reference) - self.eps
        s_minus = -(sample - self.reference) - self.eps
        self.g_plus = max(0, self.g_plus + s_plus)
        self.g_minus = max(0, self.g_minus + s_minus)
        detection = self.g_plus > self.h or self.g_minus > self.h
        votes.append(int(detection))

# majority voting
return sum(1 for elem in votes if elem == 1) >= len(votes) / 2
```

Change Detection (CD) UCB with non-stationary demand curve

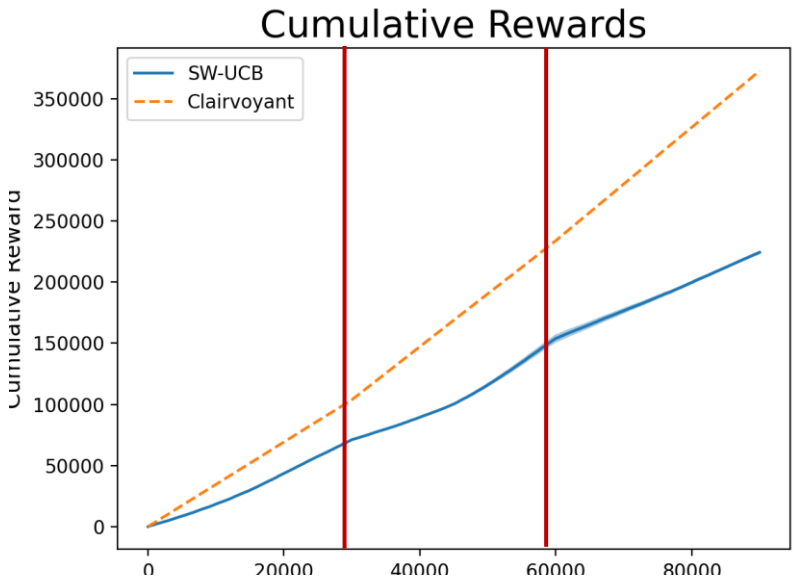


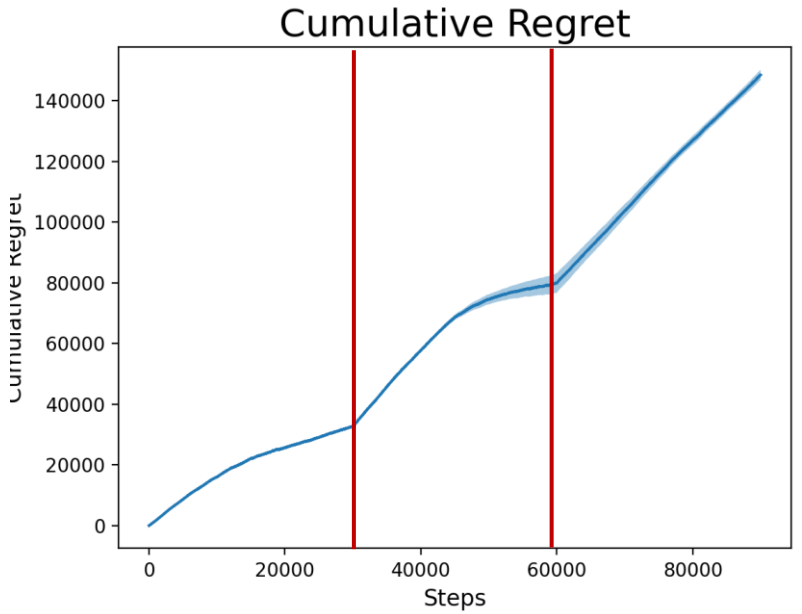


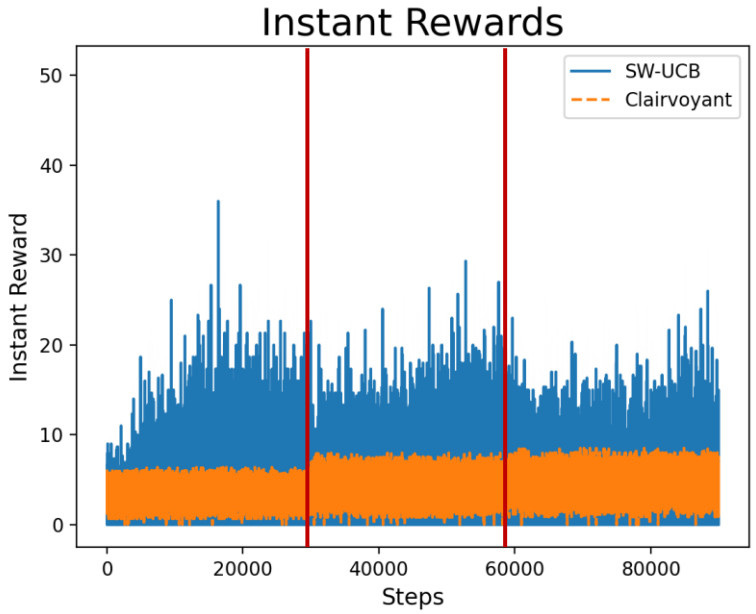


We can notice that the algorithm is able to detect the changes and with a lower delay with respect the sliding window UCB is able to pull the right arms.

UCB with non-stationary demand curve







We reported also the standard UCB (maintaining the non-stationary demand curve) where we can notice that the algorithm is not able to handle them bringing an higher regret

Context Awareness

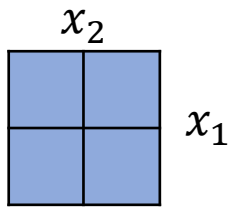
In this setup we add the consideration of user features and we use a Context Generator algorithm to decide if and how to split the features space in order to obtain different user classes and learn the demand curve of each class and exploit this information to improve the observed rewards.

Every two week the Context Generator is updated with the collected rewards and the features of the respective user. Then it evaluates wether to perform a split to form classes. If more classes are generated, a bandit for each of them is instantiated and fed with historical data regarding only data belonging to the bandit class. From now on each user will be classified and his interaction will be handled by the bandit corresponding to the user class until the new Context Generator update.

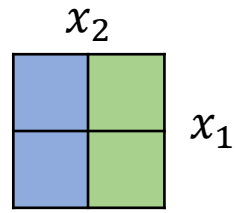
Context Awareness

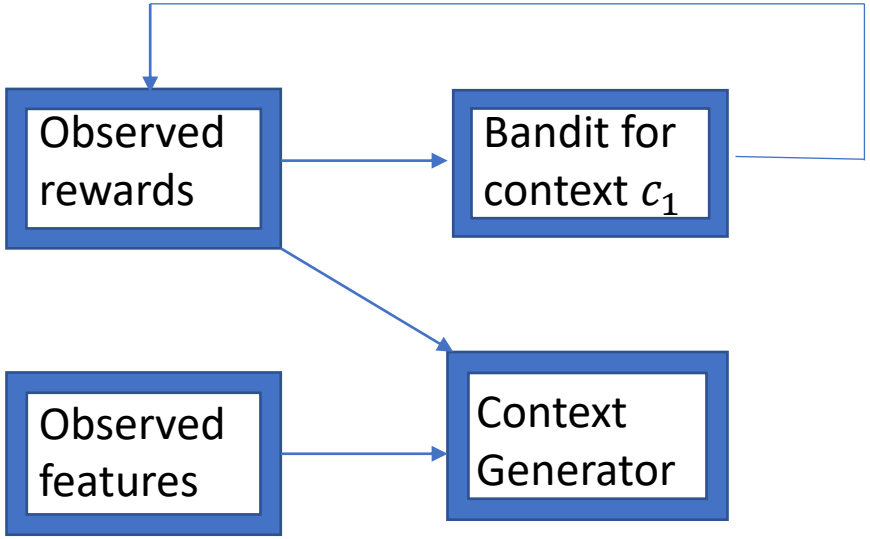
Graphical representation of an example of context handling (in our case we just have two binary features x_1, x_2):

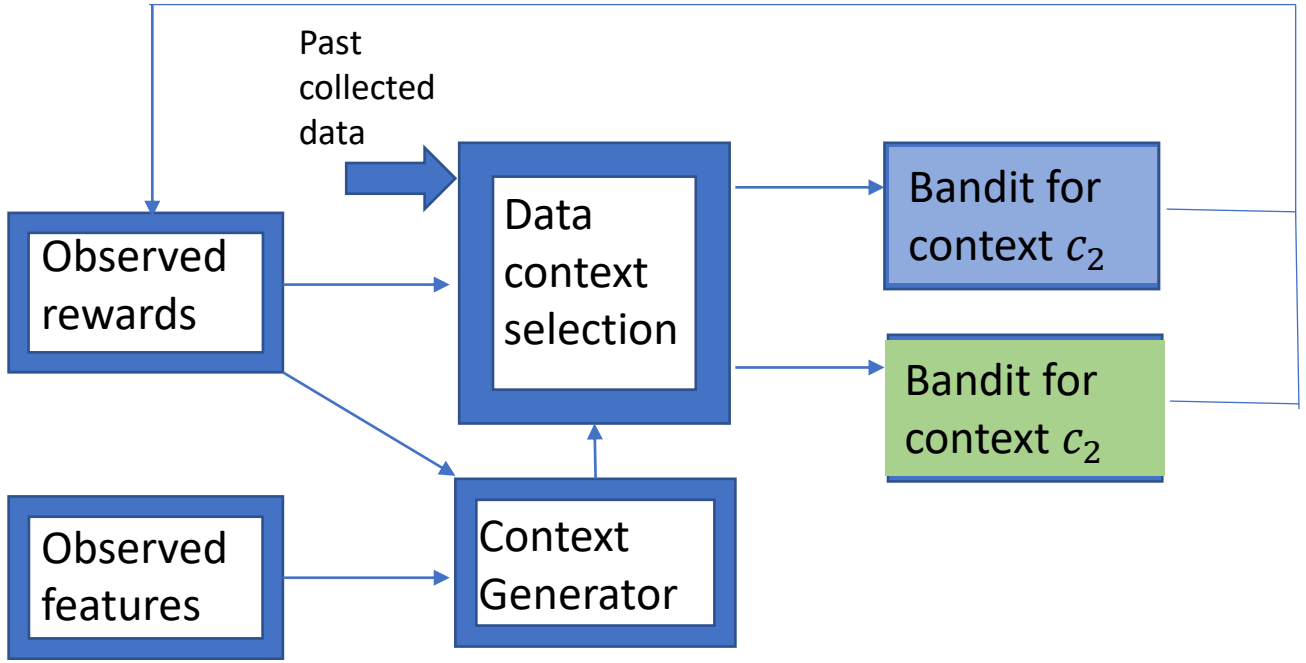
Initial context c_1 with a unique split



After 2 weeks, CG algo performs a split on features space obtaining context c_2







Context Generator algorithm

The Context Generator algorithm splits the features space by considering at each time a single feature building a binary decision tree. Each decision is made comparing two measures: the expected rewards obtainable by splitting and the ones when no split is done. This expectations are computed from past observed data and to be more restrictive over the split conditions we consider lower bounds.

The splitting condition used in this project is the following:

$$p_{c1} * \underline{\mu_{c1}} + p_{c2} * \underline{\mu_{c2}} \geq \underline{\mu_{c0}}$$

- p_{ci} = probability that context i occurs
- $\underline{\mu_{ci}}$ = lower bound of expected rewards for context i

Context Generator algorithm

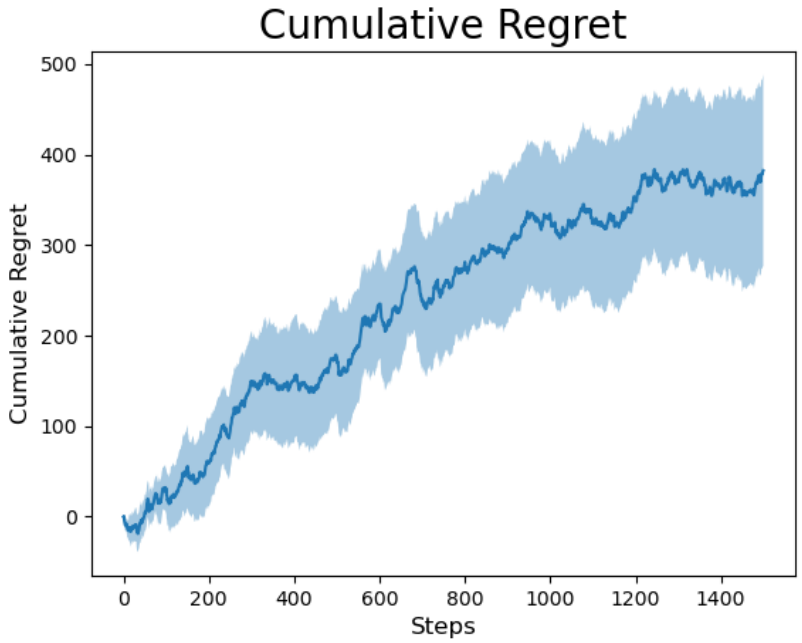
The lower bound used, valid for finite distributions, is the Hoeffding bound:

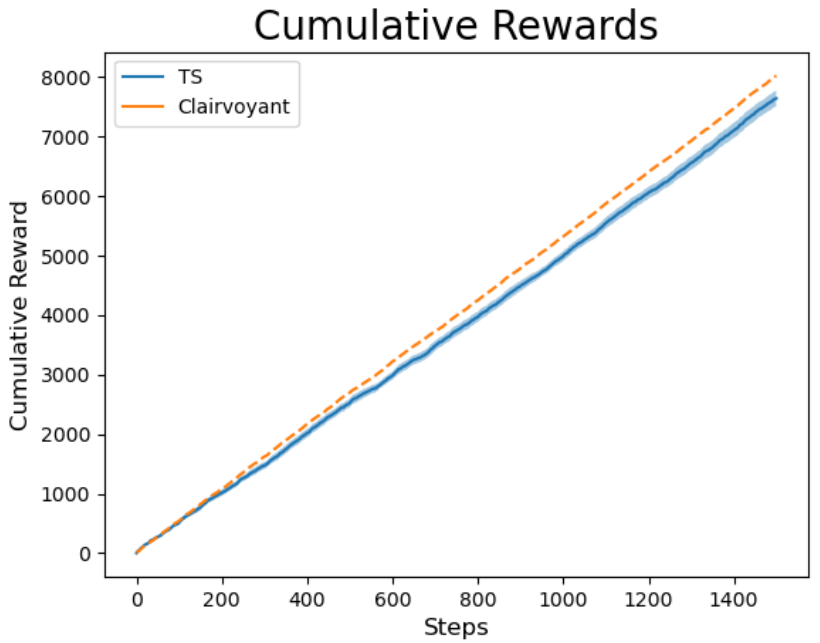
$$\bar{x} - \sqrt{-\frac{\log(\delta)}{2|Z|}}$$

- $\bar{x}$ = empirical average of rewards
- δ = confidence of the lower bound
- $|Z|$ = cardinality of the set of data considered

In this way the split in classes is performed only if evidence of benefit is found

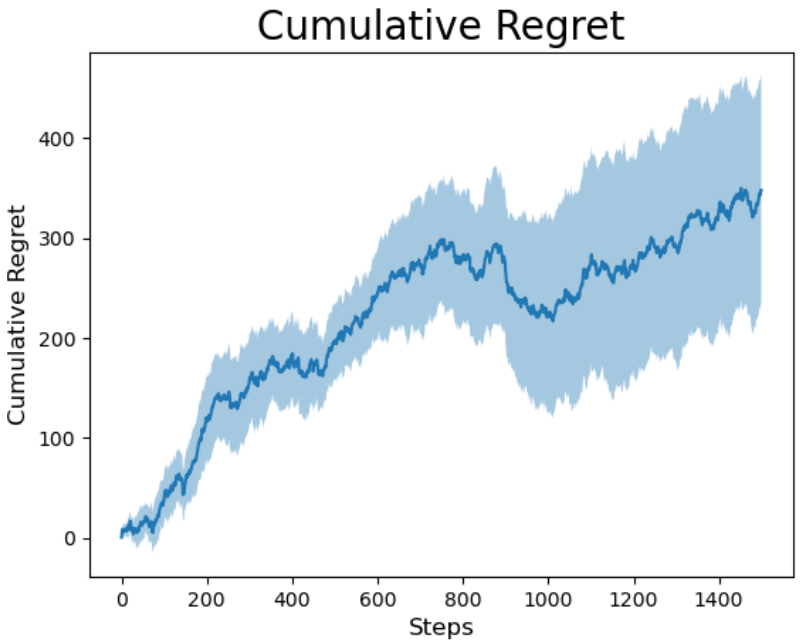
TS with uncertain conversion rates – Pull arm with Complete method

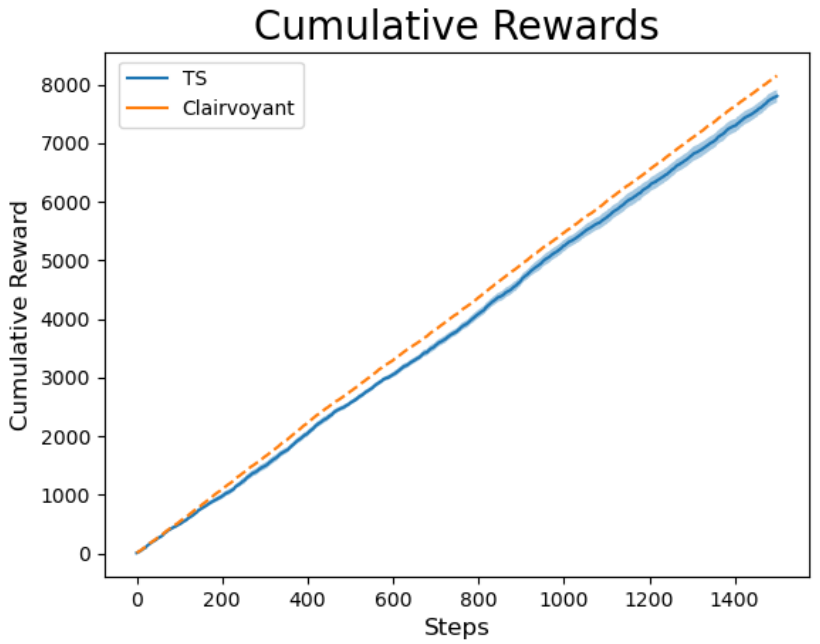




Since for the context generation we used the complete method in the bandits pull arm, we reported also the TS standard case for the comparisons.

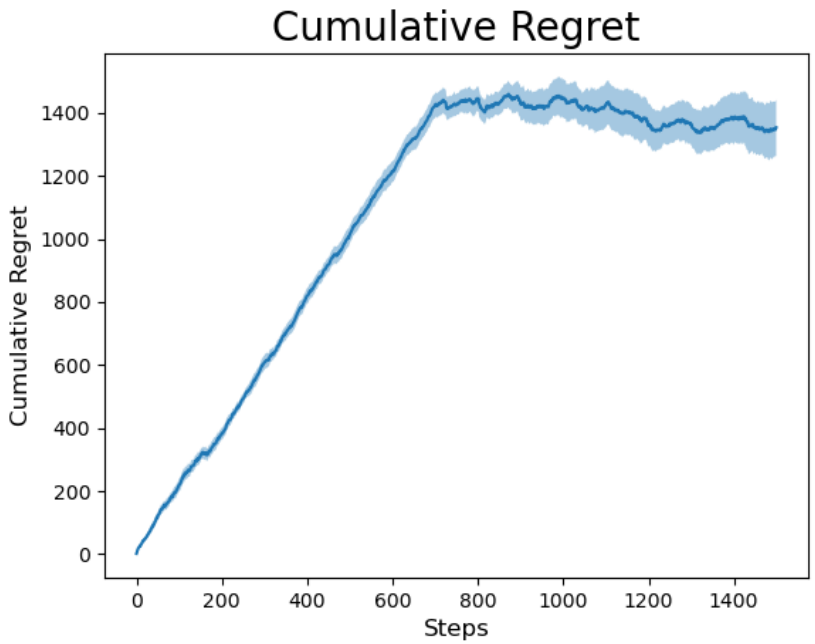
TS with context generation

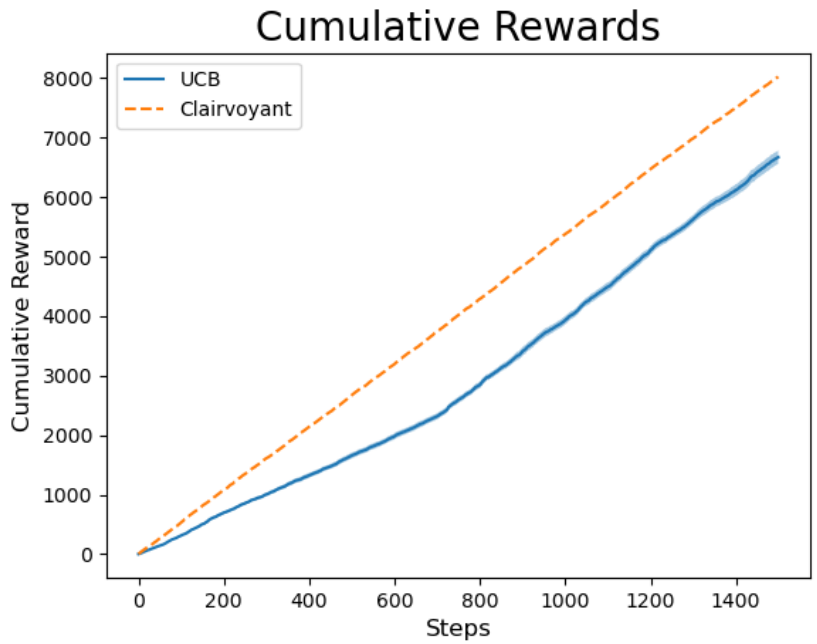




As we can see from the graphs the TS with context generation brings better results.

UCB with context generation





We can see that the UCB performs well but not good as the TS