ONLINE LEARNING APPLICATIONS PROJECT

A. Y. 2021/2022

Team Members:

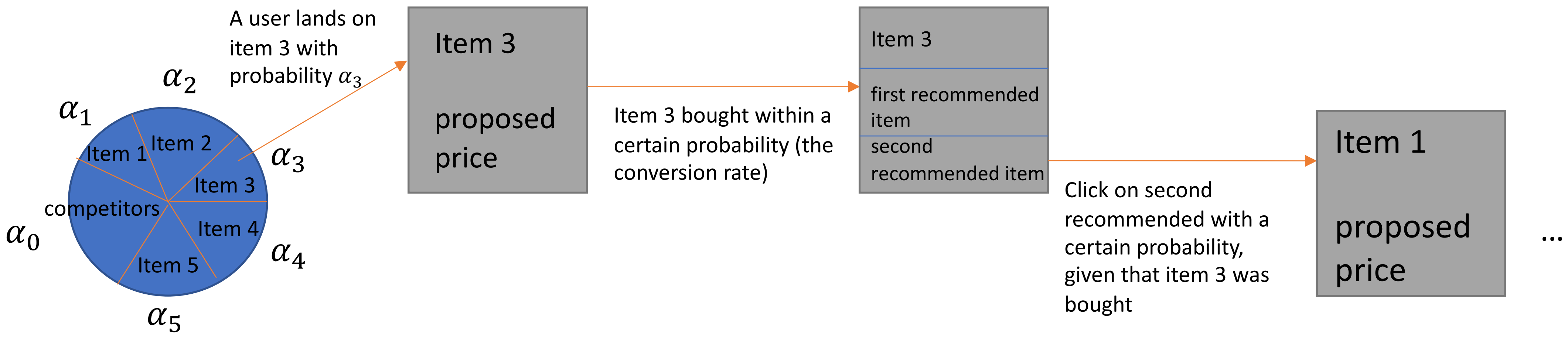
- Antonio Ercolani
- Francesco Romanò
- Andrea Lentini
- Ledio Sheshori

Topics chosen: Social influence + Pricing

The problem

- 5 Items
- 4 Prices for each item
- Prices can change once a day

- The purchase of an item influences the purchase of other items
- When an item is bought, 2 items will be recommended
- Probabilities on purchasing primary and secondary items



Optimization algorithm

- Objective function: the maximization of the **cumulative expected margin**.
- All parameters are known
- Iterative

Initial configuration: all lowest prices

Increase once at a time every item price (5 configurations for each iteration)

Evaluate which has the best marginal increase

Stop if there is no increment

- Problems of this algorithm

Probability matrix

- Determines the behaviour of a user on the website.
- Sampled randomly satisfying the primary-secondaries mapping decided by the business unit.
- The probability of clicking on the first secondary is sampled from a uniform distribution.
- While for the second element, the probability is multiplied by the lambda parameter

Alpha ratios

- Responsible for the starting page of the users
- 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

- 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.

```
n_items_to_buy_distr = np.array([[[3, 2],
                                  [6, 2],
                                  [2, 2],
                                  [7, 2],
                                  [4, 2]],
```

5 products, 1 distr. per product

Mean of the distr.
=
Average number
of sold items

Variance

Simulator

The following pseudocode shows a 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, collect all the rewards that he generates during the purchases on the website
- Compute the regret by comparing it with the estimated optimum reward

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

- For each item, consider it as a seed and simulate a random walk
- Estimate activation probabilities from the considered seed *i* to an item *j* as the ratio:

  #times j is visited from seed i / #simulations

- The result is the matrix with all the activation probabilities

Optimum computation

We build a matrix *opt_per_starting_point* 5x3 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 matrix is built 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
$A_{x\text{->}y}$: 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($P_1*C_1*N_1 + A_{1\text{->}2} *P_2*C_2*N_2 + A_{1\text{->}3}\ P_3*C_3*N_3 + A_{1\text{->}4} *P_4*C_4*N_4 + A_{1\text{->}5} *P_5*C_5*N_5$)

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

EXAMPLE: if a user of class 1 starting from item 2 generates a reward of 75 during his visit the associated regred will be:
regret = opt_per_starting_point[2][1] - 75 = 100 - 75 = 25

Bandit - Pull arm

The behavior of the user in the graph (the purchase of a series of products) is regulated by 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:

Item 1 **Higher price** **Lower conversion rate** → Due to the low conversion rate of item 1 the user **will rarely buy other items** → **Lower user reward**

Item 1 **Lower price** **Higher conversion rate** → Item 2 **High reward** → Item 3 **High reward** → **Higher user reward**

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.

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):

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

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
$A_{x\text{->}y}$: 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 developed two similar algorithm to face the problem:

Complete method

Evaluation of every combination

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

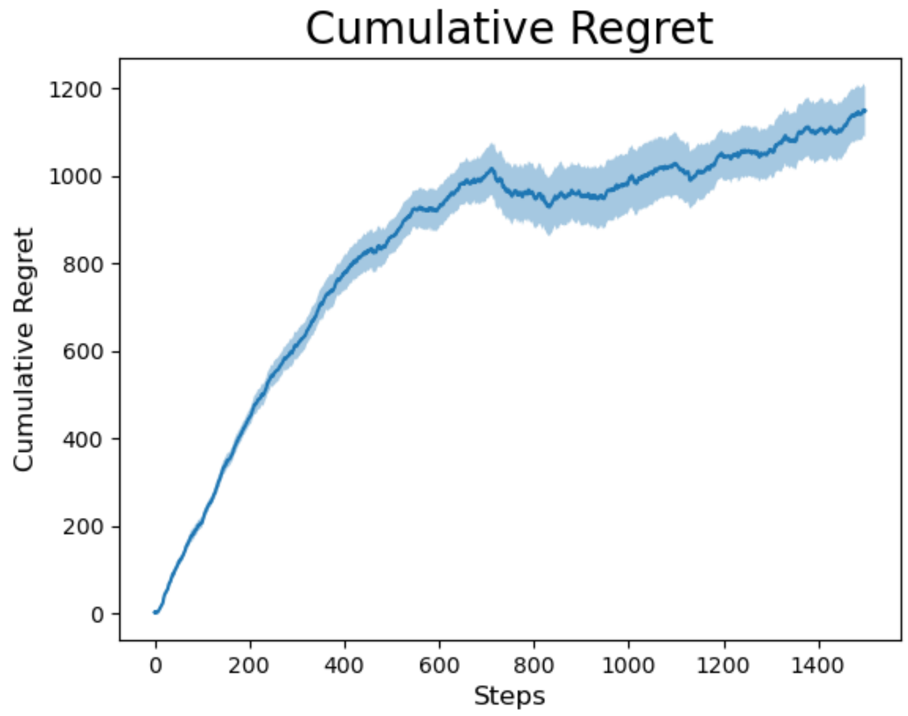
Estimated method

Keeps the arms pulled the day before changing only 1 arm at the time.

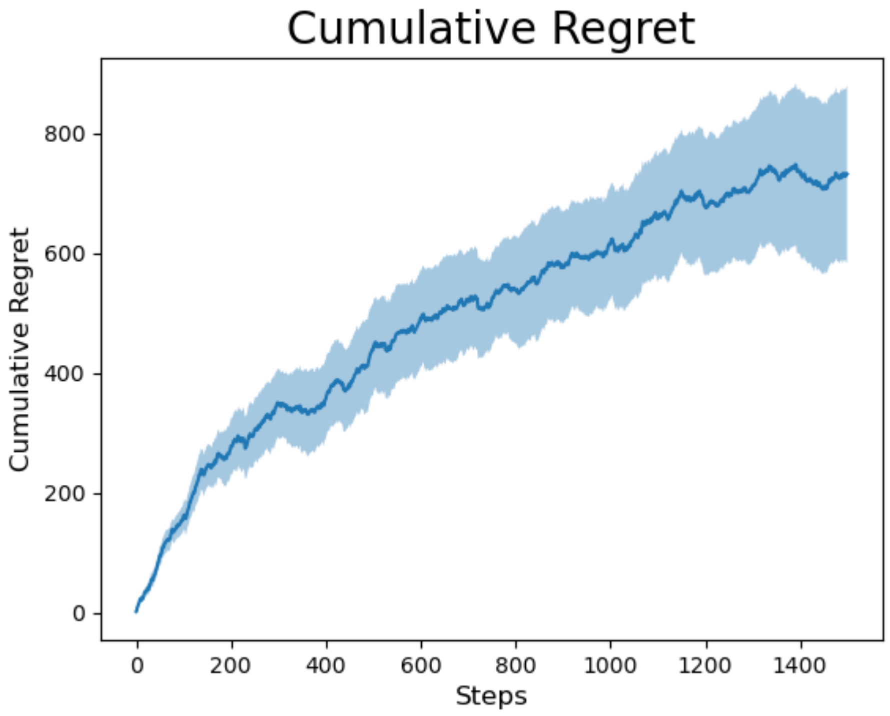
Number of combinations: 5 * 4 * 5 = 100

CONS: can change only 1 arm per day

Uncertain conversion rates



UCB



TS

Both bandits are able to find the arms that lower the regret, moreover as we expect from theory the TS algorithm reaches a lower regret

Uncertain α ratios, and number of items sold per product

Uncertain number of items sold per product

↓

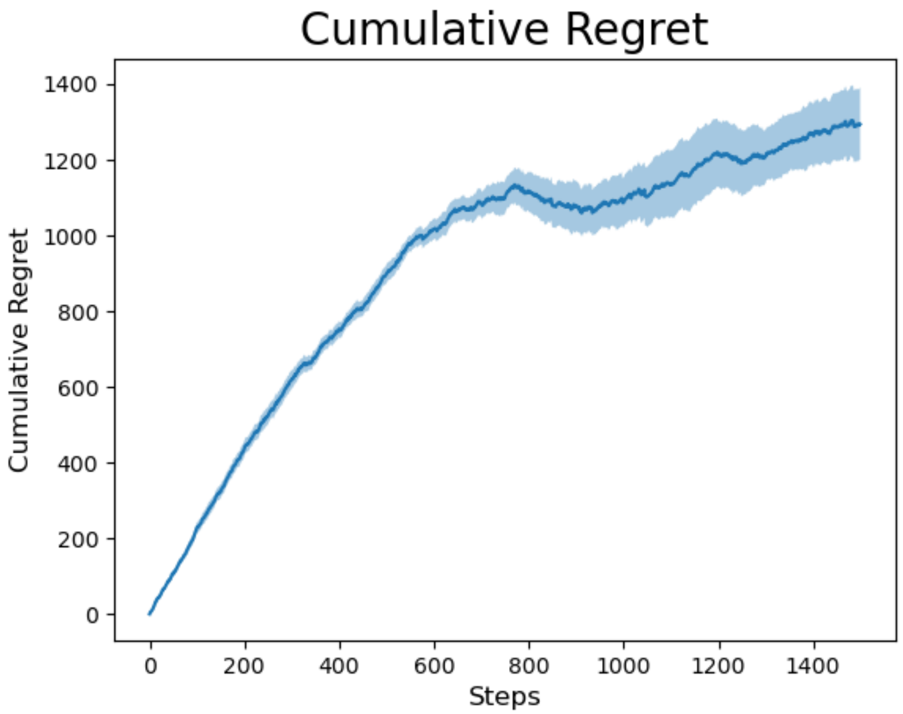
Simple **estimation averaging the observed outcomes**, discriminating per user class

Uncertain α ratios

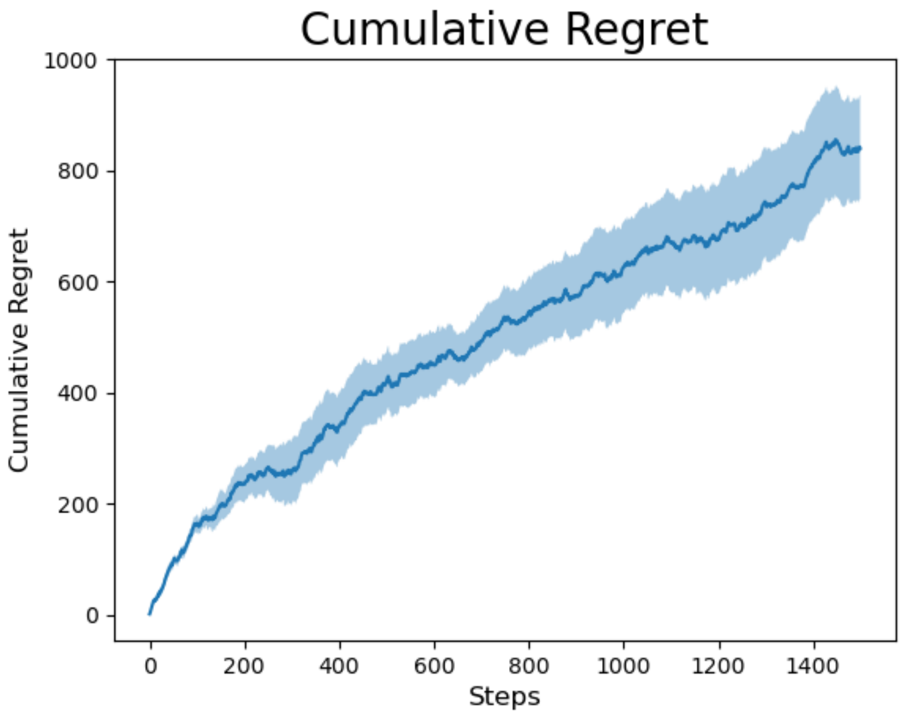
↓

Nothing to do since only the environment uses them

Uncertain number of items sold per product



UCB



TS

The estimation 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)

Probability matrix estimation

Probability matrix inference through generation of episodes and credit assignment.

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

Threaded version to speed up the computation.

Probability matrix estimation

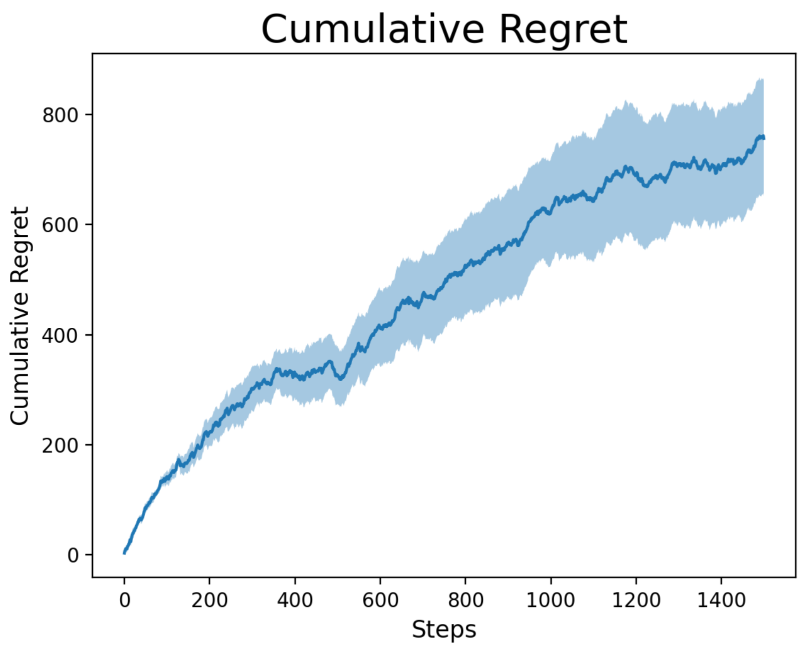
Start from a random starting node (item) and then simulate an **episode** (interaction). Variable *history*: activated nodes in the diffusion phase.

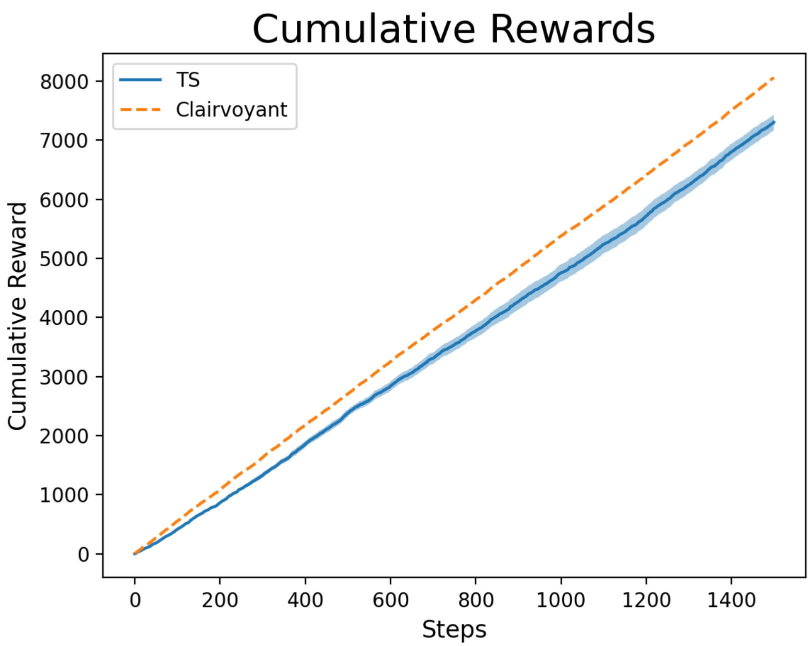
Credit assignment technique to compute final estimation for each edge.

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

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

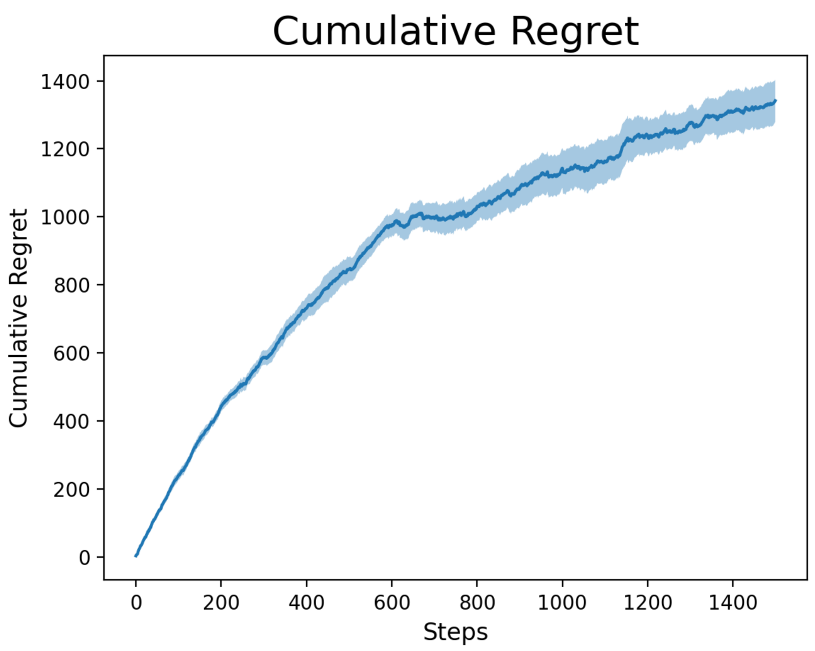
TS with uncertain graph weights

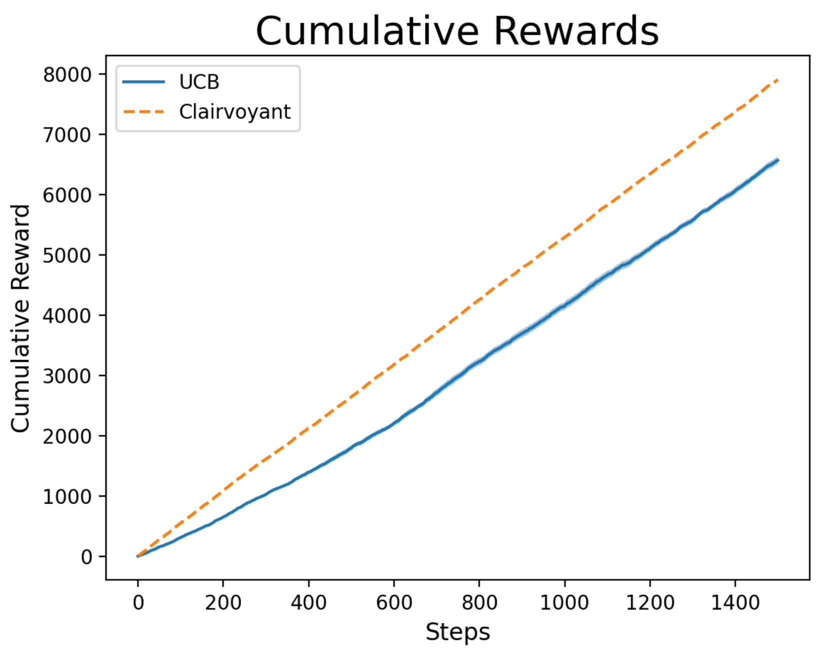




Almost same regret of the one with certain probability matrix

UCB with uncertain graph weights





The same holds for the UCB case

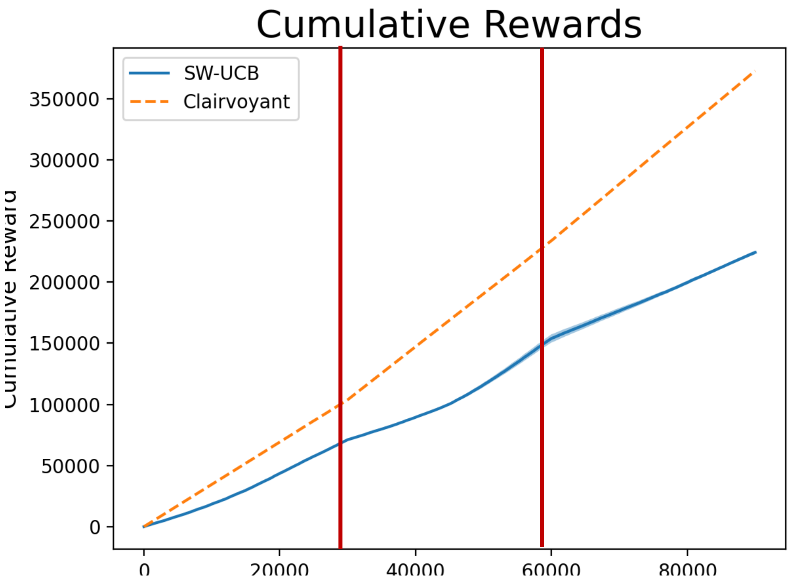
Non-stationary environment

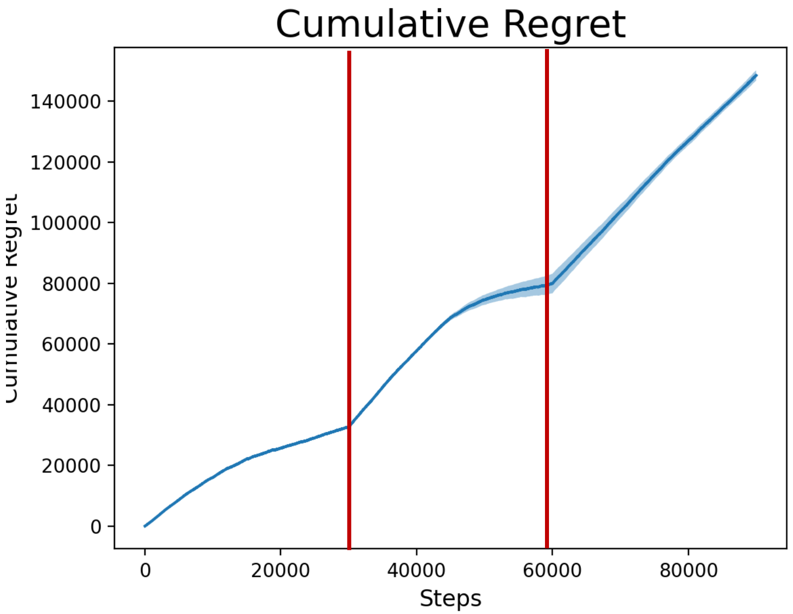
Change in the demand curves (values of the conversion rates) → new dimension in conv matrix. Three different **phases**, so modelling two **abrupt changes**.

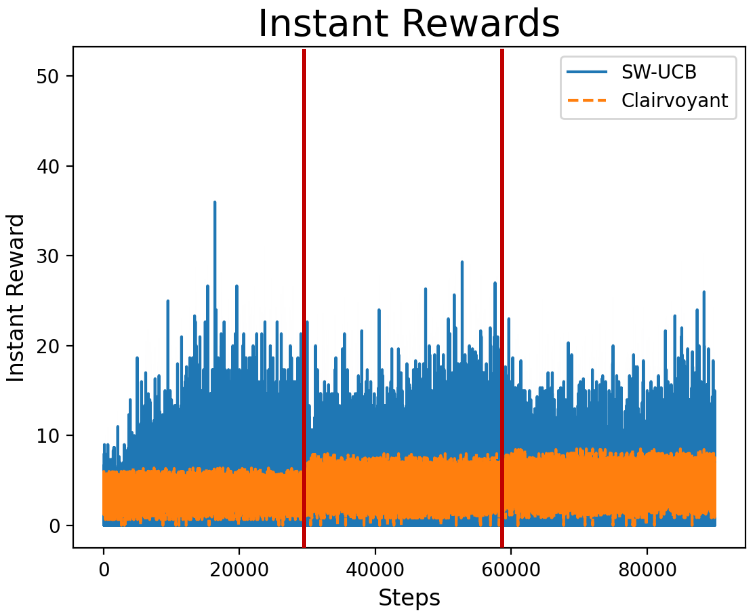
Two adapted UCB bandits:

- **Sliding-window**
- **Change detection**

UCB with non-stationary demand curve







Standard UCB not able to adapt to the changes.

Non-stationary environment: SW-UCB

Fixed-size memory bandit (last τ collected rewards).

Many users in the system every day

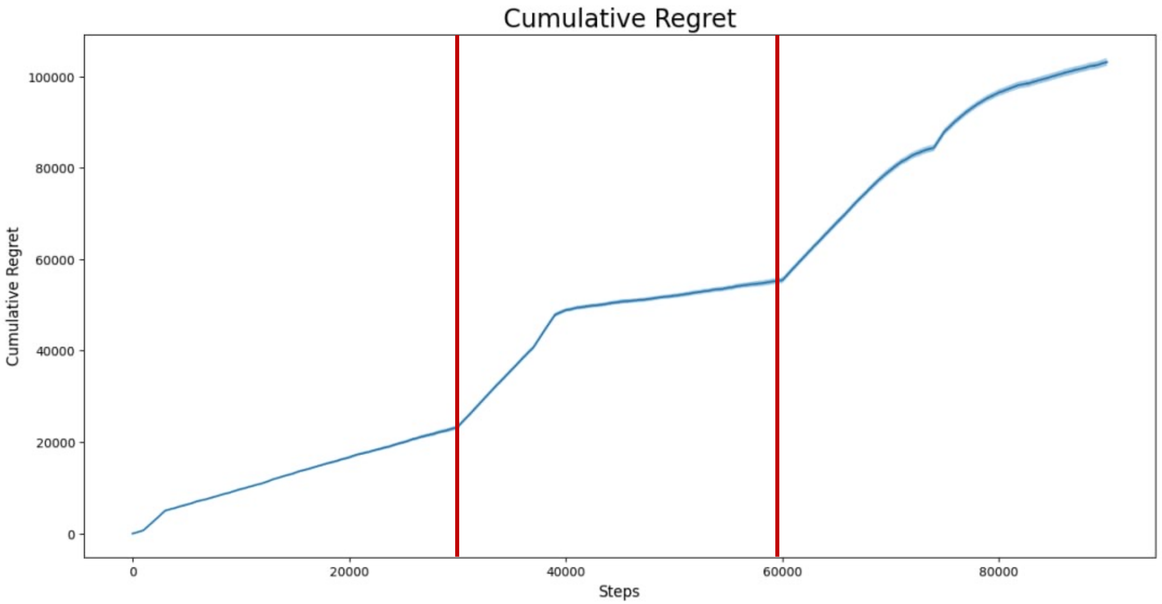
↓

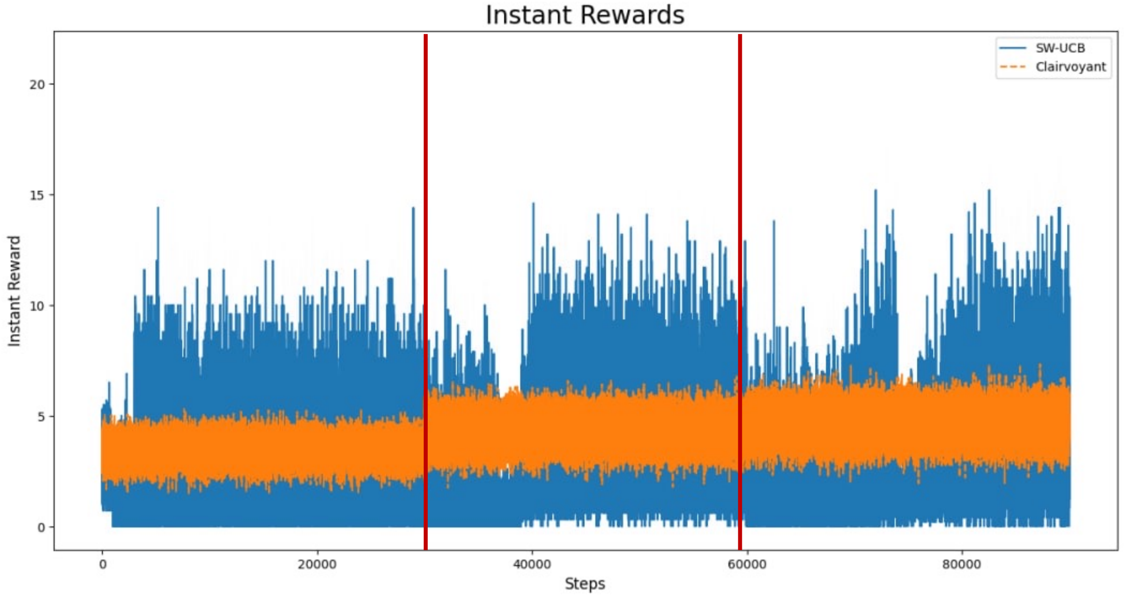
Multiple rewards (= in variable size) each day

↓

Trick with **placeholders**

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





Bandit able to adapt to the changes after a certain delay bringing a lower regret each time.

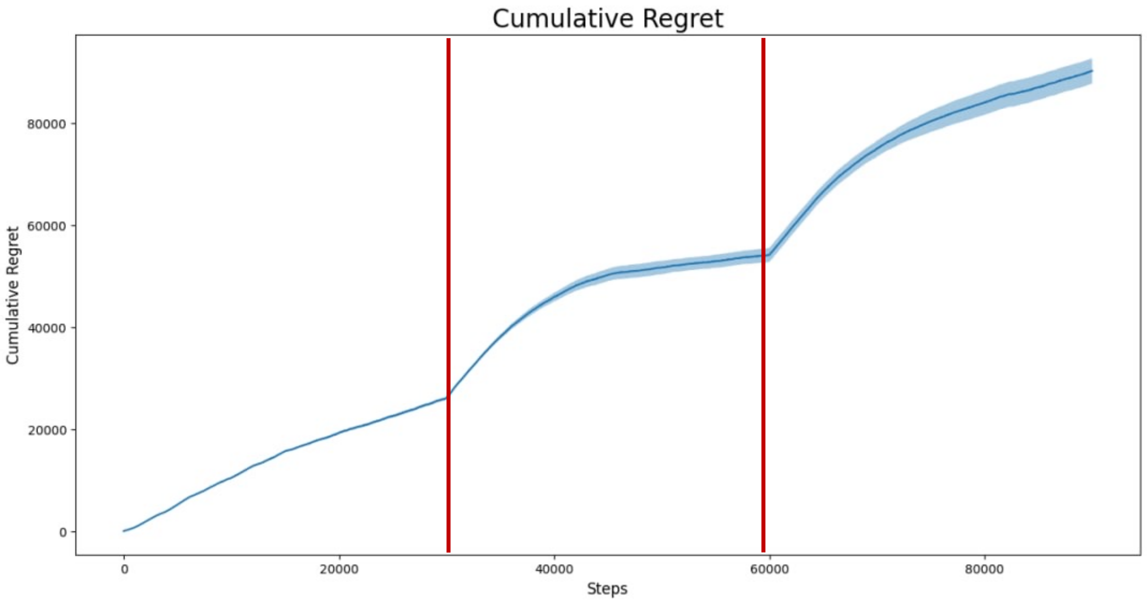
Non-stationary environment: CD-UCB

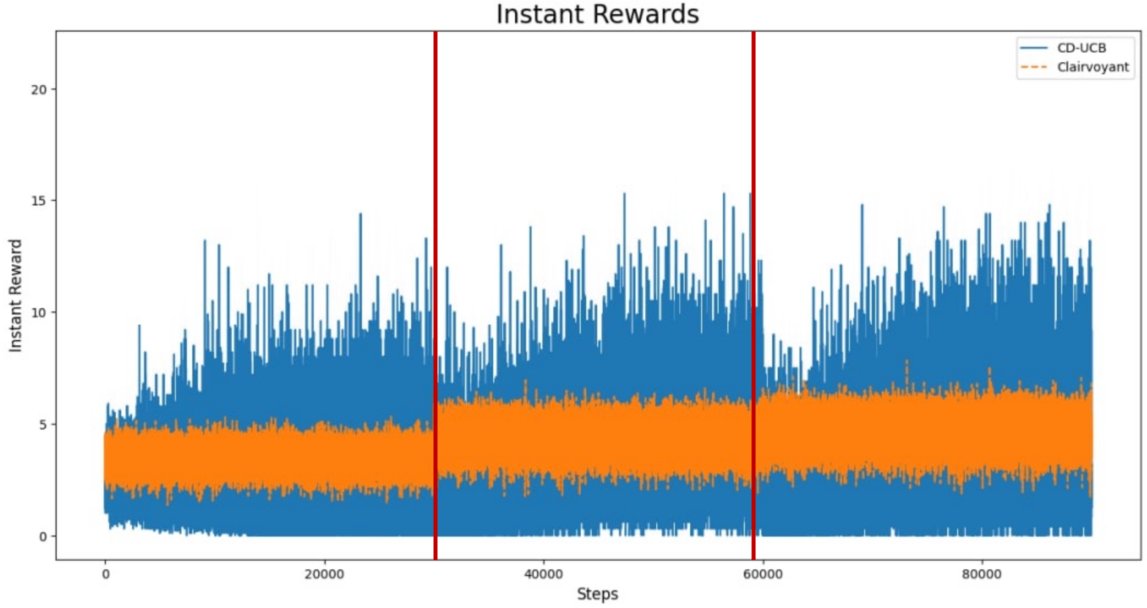
Memory controlled by change dection mechanism.

If a change is detected for a certain arm for an item, swipe the collected rewards for that configuration.

CUSUM change detection algorithm, adapted for multiple rewards through **majority voting**.

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





Lower delay with respect to sliding window UCB to detect the changes.

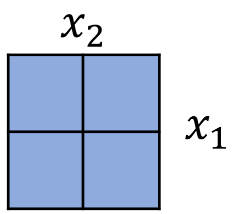
Context Awareness

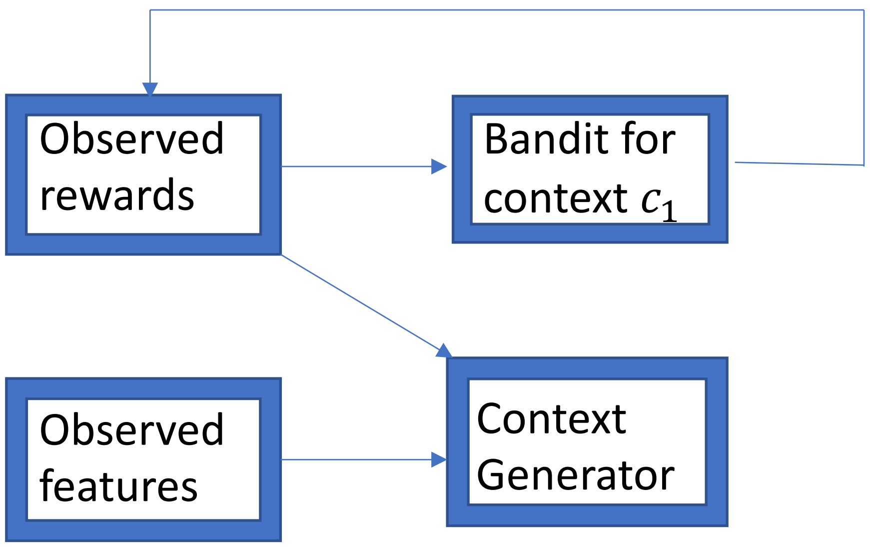
- Add the consideration of user features to **classify users**, in order to learn class specific demand curves and improve the observed rewards
- A Context Generator (**CG**) algorithm periodically **builds a decision tree**, by splitting the features space, to classify the users
- Every 2 weeks CG **evaluates if and how to split**, by estimating the split advantages/downsides
- Each split will have a **dedicated bandit** algorithm used exclusively for the split-defined user class
- The split-specific bandit is **fed with historical data**, and then updated only with observations of its class

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

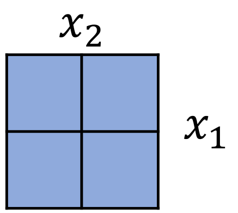




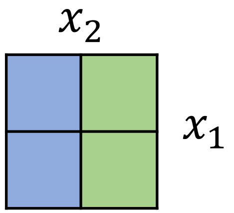
Context Awareness

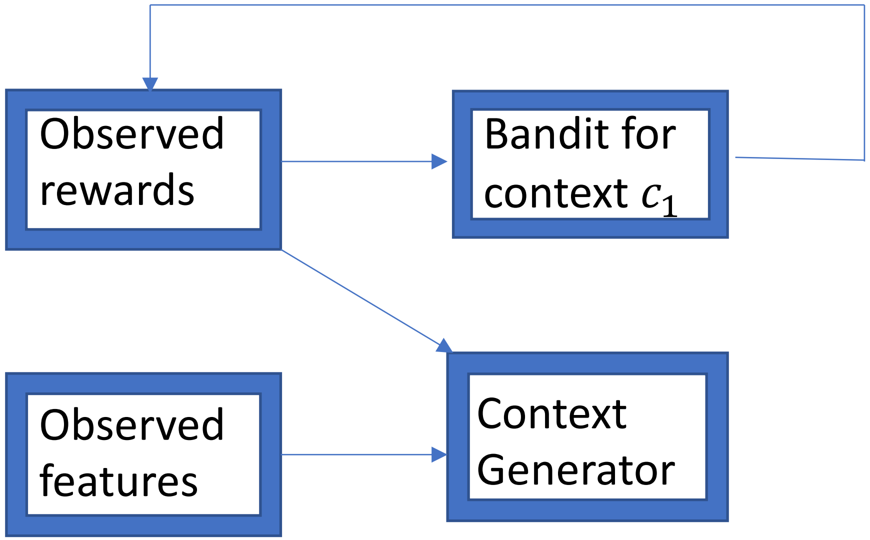
Graphical representation of an example of context handling (in our case we just have two binary features x_1, x_2):

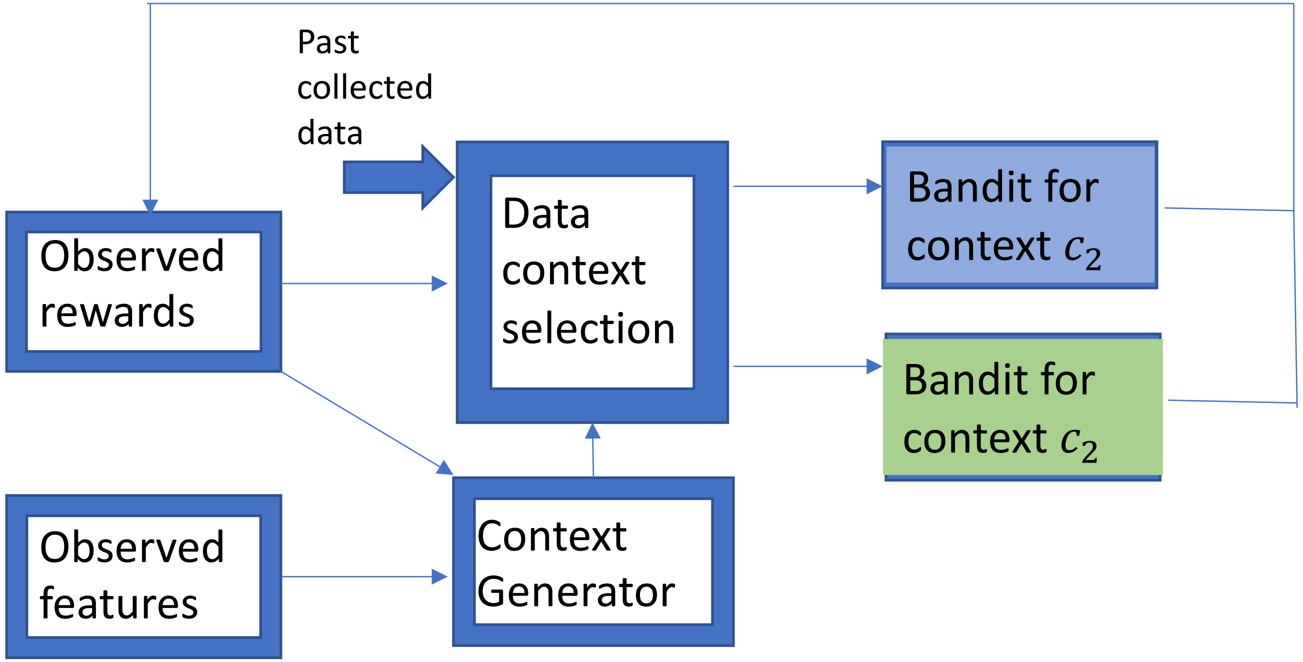
Initial context c_1 with a unique split



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

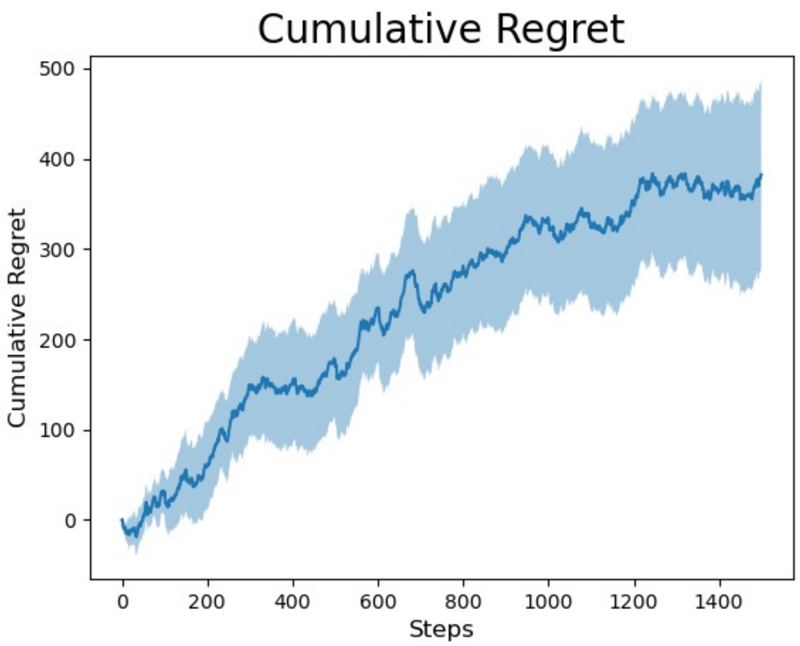
The lower bound used, valid for finite distributions, is the Hoeffding bound:

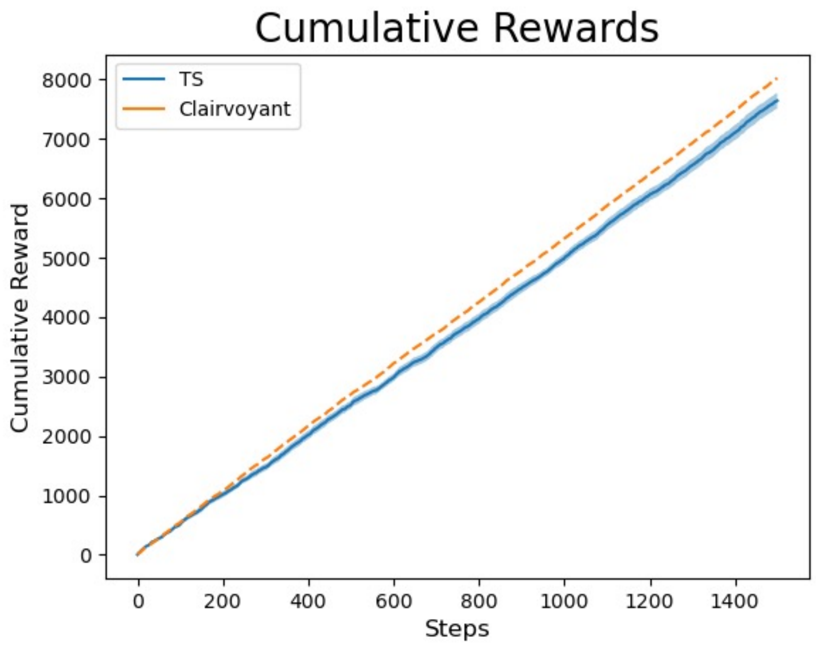
$$\bar{x} - \sqrt{-\frac{\log(\delta)}{2|Z|}}$$

- $\bar{x}$ = 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

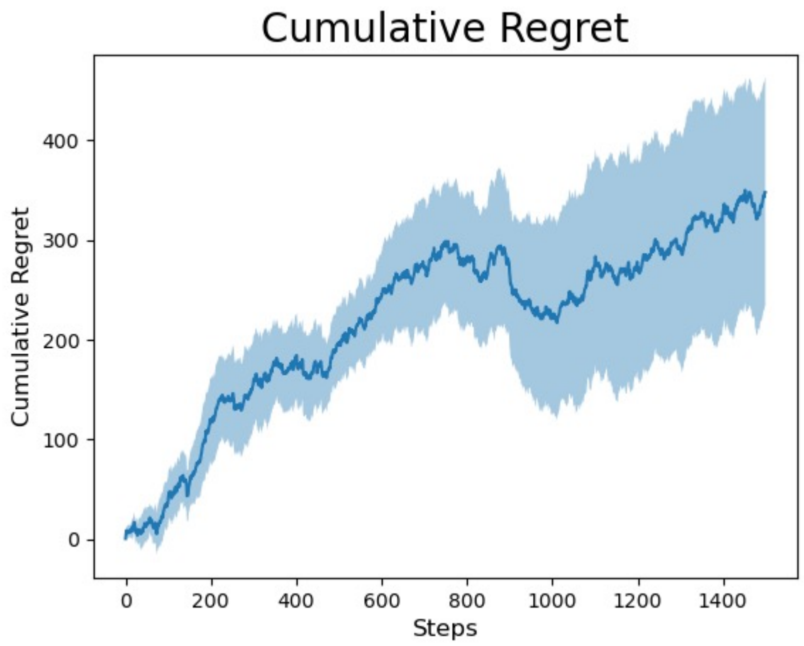
TS with uncertain conversion rates – Pull arm with Complete method

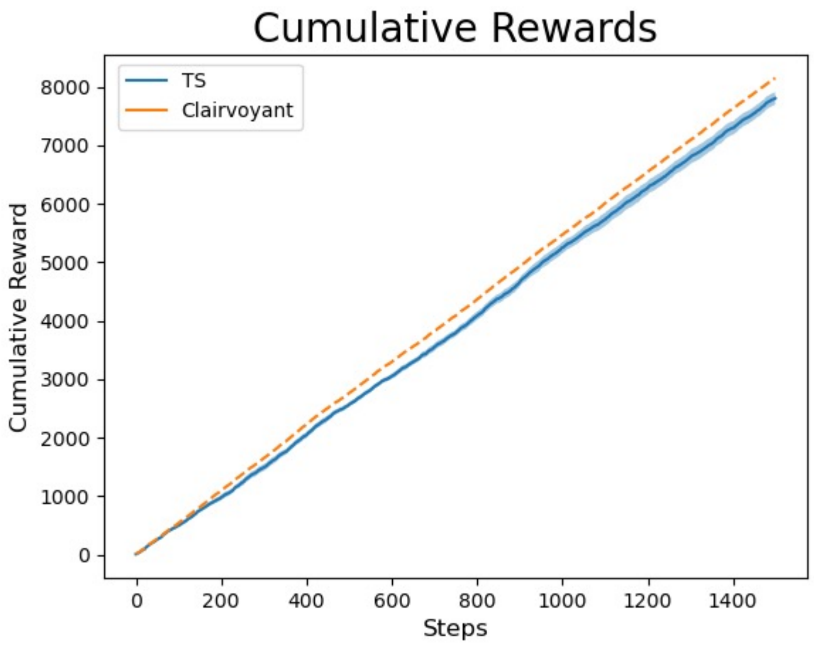




Since for the context generation we used the complete method in the bandits pull arm, we reported also the TS standard case for the comparisons.

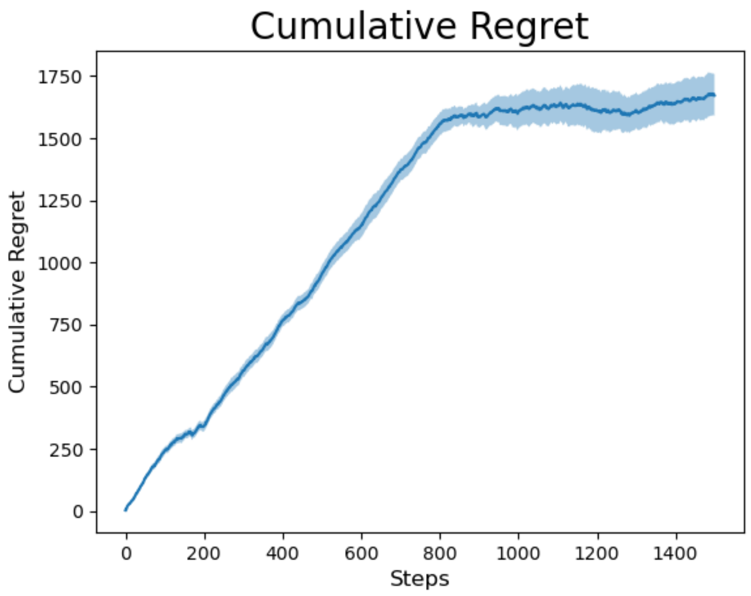
TS with context generation



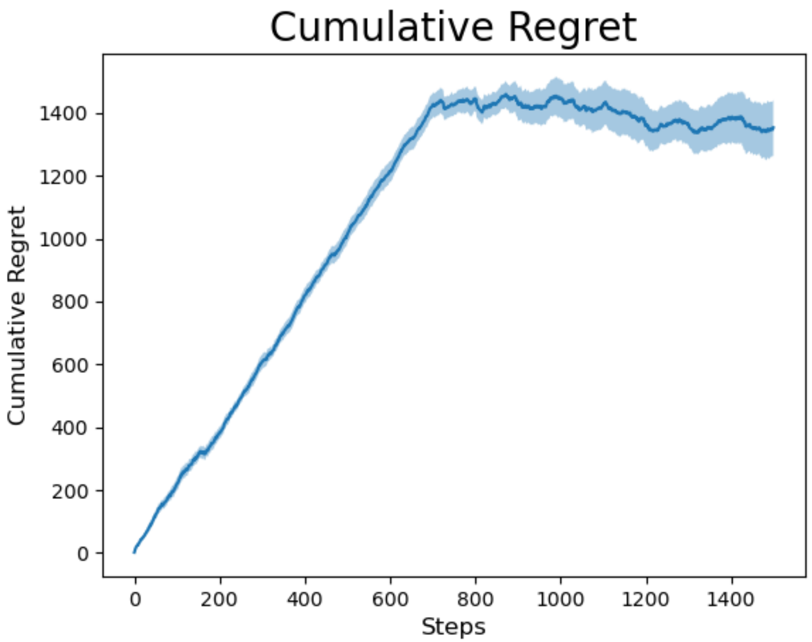


As we can see from the graphs the TS with context generation brings better results.

UCB with context generation



Without context



With context

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