Project Report

Pricing and Social Influence

Online Learning Applications

2021/2022

Alessandro Immordino (10803969)
Giacomo Mosca (10574012)
Riccardo Pala (10805584)
Giacomo Polvanesi (10806282)

Prof. **Nicola Gatti**

POLITECNICO
MILANO 1863

POLITECNICO
MILANO 1863

Step 1

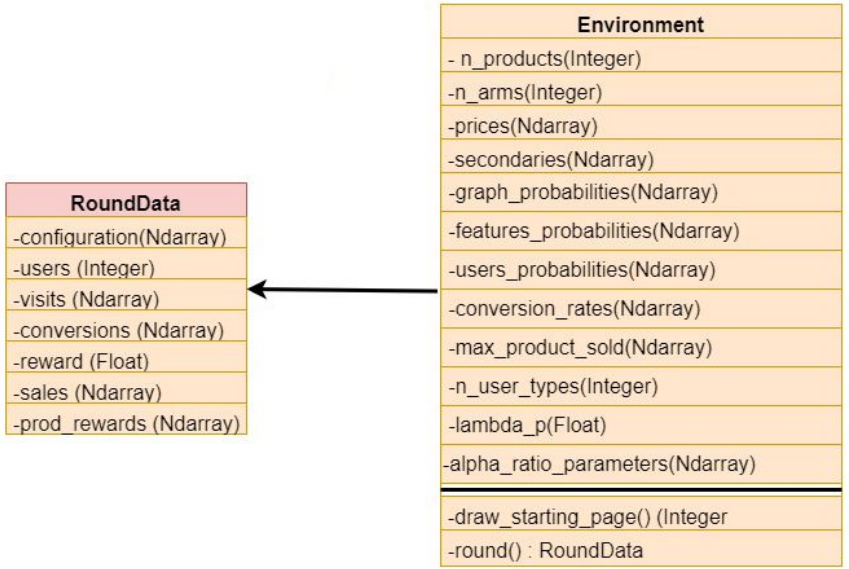
Environment

Environment

The simulator for the problem described in the specifics is developed through the **Environment** class. The class holds all the probabilities and parameters necessary to model the users' behavior and can simulate the sales obtained during a single day.
No supplementary classes are used to represent the state of the simulations in order to keep the implementation as streamlined as possible.

For each day of the simulation, the Environment can be called to return the **total reward** gained by the business in question from the users' purchases.
All the information about a day's run that will be visible to the optimization algorithm is also collected in a **RoundData** object, which will be passed to the learner at the end of each simulation.

As per specification, the Environment keeps track of the sales of 5 products, each of which can have 4 different possible prices.

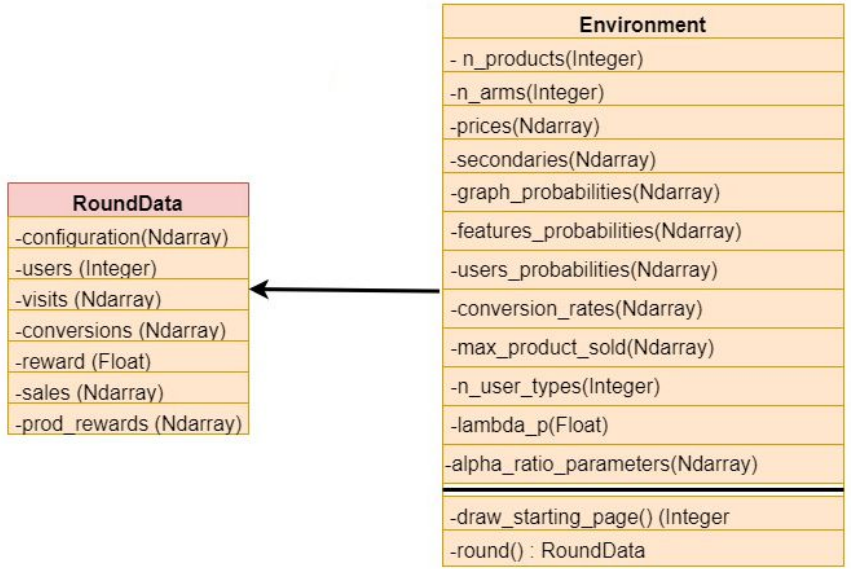


User classes

To model a day of sales, we first need to specify the different **user classes**. There are 3 types of users present in the simulation, defined by 2 binary features as such:

feature 1 \ *feature 2*	***False***	***True***
False	*Class 0*	*Class 0*
True	*Class 1*	*Class 2*

Users from different classes have different behaviors, so all the probabilities and distributions defined in the Environment have **3 sets of values** that model a different user class each.

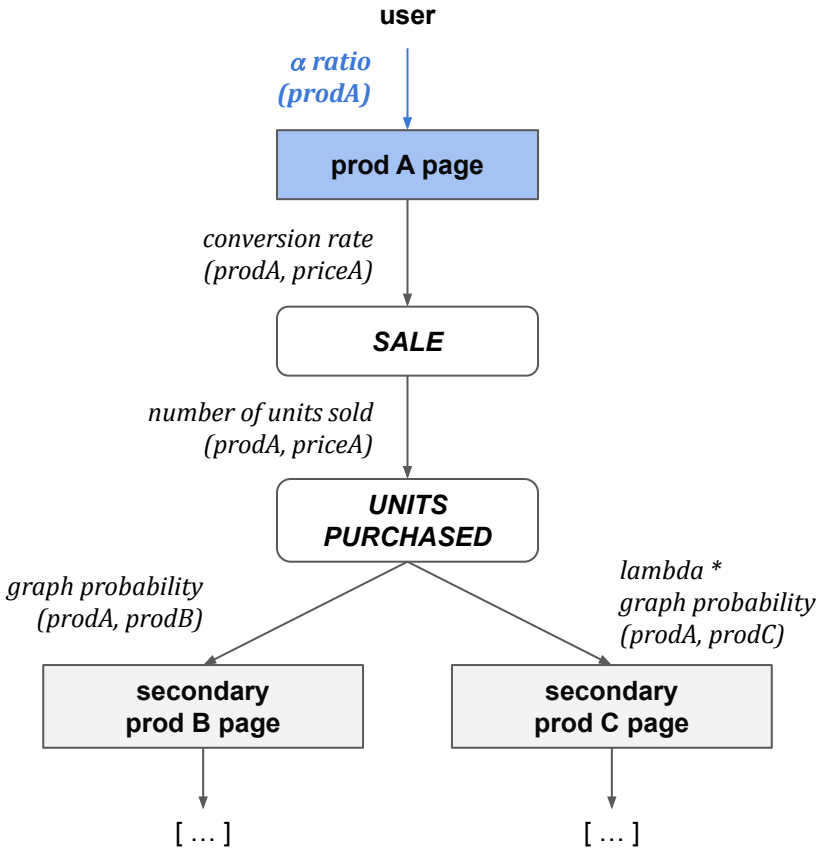


During each simulation a random number of users is chosen for the day. Whenever a new user enters the simulation, their behavior is modelled following the **decision tree** represented here.
As stated before new users are also immediately assigned to their respective class: all the values considered from here are also a function of the user's class.

Users are first randomly assigned to their starting page according to the product's landing probability or α **ratio**.
The α ratios are modelled in the Environment by a set of independent **Beta distributions**, one for each product plus one for users landing on a competitor's page.
At the beginning of each round these distributions are sampled and the resulting vector of values is normalized to obtain the landing probabilities for the round.

Note that whenever average results needed to be calculated with these α ratios their mean value was used, which can simply be computed from the distribution's parameters as:

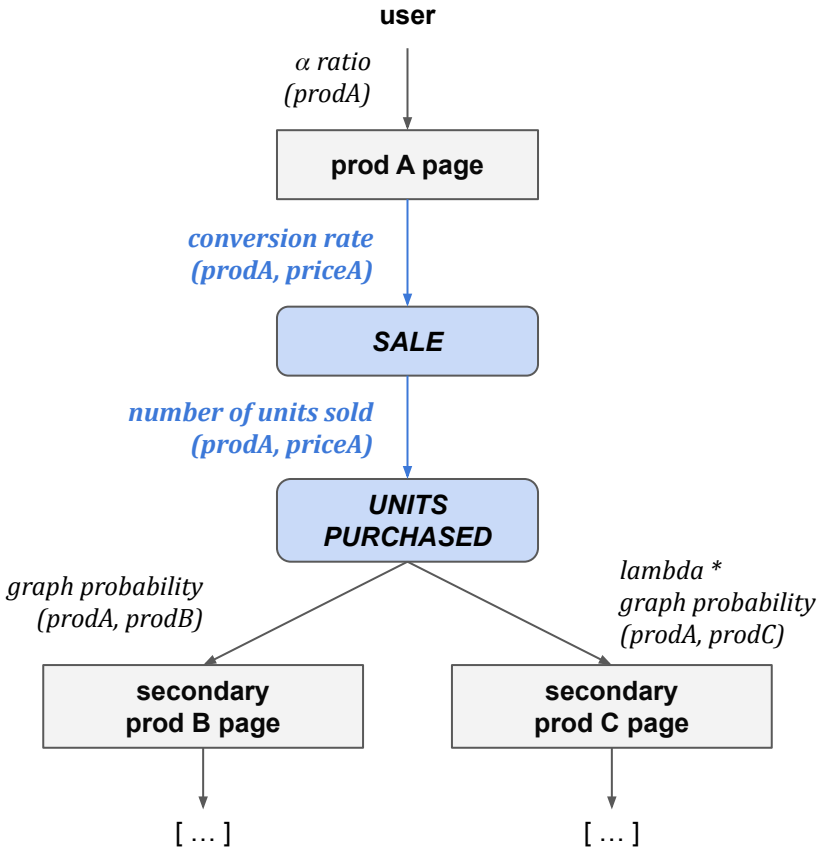
$$\mu = 1 / (1 + \beta / \alpha)$$



After the user's starting product page is chosen, a sale can be made according to the product's **conversion rate**. The conversion rate is modelled as a binomial distribution with set mean, with different values depending both on the product in question and its current price.
Note that the choice of the conversion rates is used to model the user's **reservation price** as well: if the price of the product falls above the user's reservation price for that product, its conversion rate is set to 0.

If a sale is made, the **number of units purchased** can then be randomly picked. This is chosen as a random value between 1 and the maximum number of units purchasable by the user, which is once again a function of the product and its price.

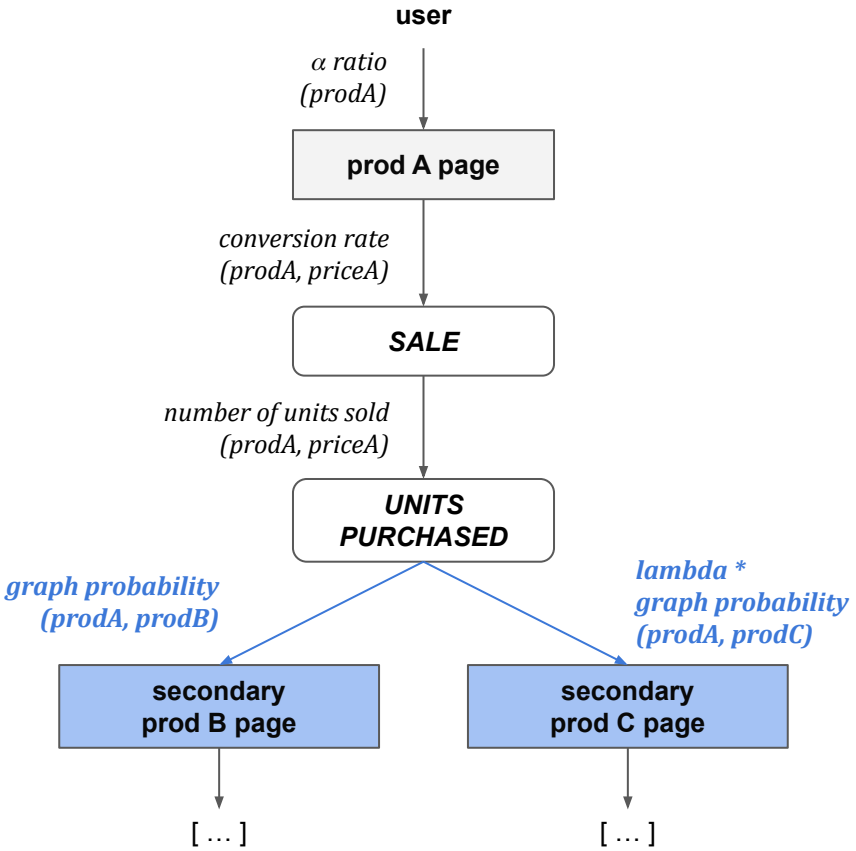
The product between the number of units purchased and the product's price represents the reward obtained by the business for this single interaction.



In case of a successful sale, the user is also shown an ordered list of two suggested or **secondary products** that they can independently click on to continue their purchases.

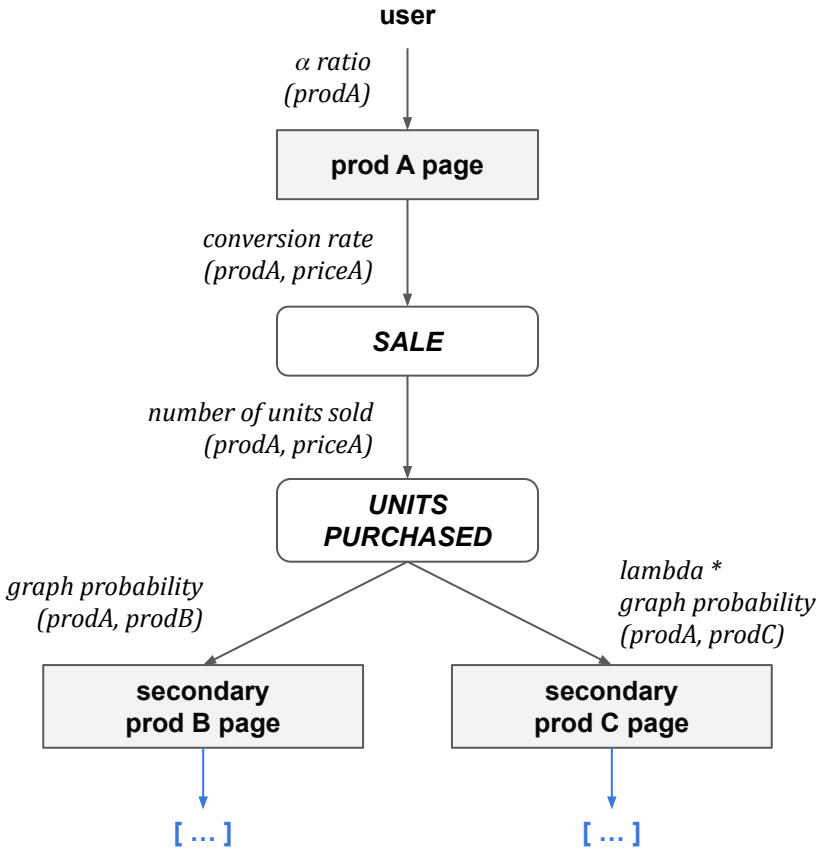
The **graph probabilities** represent the probability that the user will click on the secondary product's page when the recommendation is shown. These are also modelled as binomial distributions with known mean, one for each primary-secondary product pair.
For secondary products displayed in the second recommended slot, their graph probability is also multiplied by a known **lambda** factor to obtain the true probability of accessing the page.

The list and order of secondary products displayed for each primary is fixed: therefore, the graph probabilities between products that can't appear as secondaries are effectively 0. However, the implementation of the Environment class allows to easily lift this restriction for a more general case and set as many secondary products as desired through these graph probabilities.



If a secondary page is reached, the process now continues from the top, with the exception that a previously visited will never be clicked again. This is simply implemented by keeping a record of each user's visits and preventing any repeats.

Whenever one of these binomial checks is failed the user ends their actions on this branch of their decision tree, and when all actions are exhausted their visit ends. The **final reward** obtained from all their purchases is added up to the total reward for the day.



POLITECNICO
MILANO 1863

Step 2

Optimization algorithm

Greedy algorithm

An initial greedy approach to solving the optimization problem is shown here, following the given specifications.

This **greedy algorithm** is implemented by a **Learner** class, which is created with full knowledge of the Environment, and operates using the following loop:

1. the algorithm starts from the configuration where all prices are set to the lowest value and evaluates its associated reward;
2. each price is individually raised by one in order to create 5 new configurations that only differ from the original by 1 element;
3. the rewards of the new configurations are evaluated;
4. the configuration with the highest reward that improves upon the starting one is picked as the starting point for the next iteration;

- if no improvement is made, the algorithm returns the current configuration.

```
def greedy_optimization(self):

    iteration = 0
    current_configuration = [0] * self.n_products

    best_configuration = current_configuration
    best_reward = self.evaluate_configuration(current_configuration)

    while any(x < (self.n_arms - 1) for x in current_configuration):
        for i in range(self.n_products):
            new_configuration = current_configuration.copy()
            if new_configuration[i] >= (self.n_arms - 1):
                continue
            new_configuration[i] += 1
            reward = self.evaluate_configuration(new_configuration)
            if reward > best_reward:
                best_configuration = new_configuration
                best_reward = reward
        if np.array_equal(current_configuration, best_configuration):
            break
        current_configuration = best_configuration
        iteration += 1
        print()

    return best_configuration
```

Greedy algorithm

Because all of the Environment's parameters are known to the algorithm, the reward of each configuration can be evaluated empirically through **simulations** of the Environment.
A configuration's reward is given by the average reward obtained over a simulated round of 10000 users, which is executed in the exact same way as the Environment's simulations.

This greedy algorithm is clearly **suboptimal** and has obvious limitations that make the results far from ideal.
Because the computed rewards depend on random simulations, the algorithm is non-deterministic and will return different configurations at each run.
On top of that, the greedy approach means that optimal configurations are often never explored depending on the Environment's values, getting stuck in any local maxima.

```
def greedy_optimization(self):

    iteration = 0
    current_configuration = [0] * self.n_products

    best_configuration = current_configuration
    best_reward = self.evaluate_configuration(current_configuration)

    while any(x < (self.n_arms - 1) for x in current_configuration):
        for i in range(self.n_products):
            new_configuration = current_configuration.copy()
            if new_configuration[i] >= (self.n_arms - 1):
                continue
            new_configuration[i] += 1
            reward = self.evaluate_configuration(new_configuration)
            if reward > best_reward:
                best_configuration = new_configuration
                best_reward = reward
        if np.array_equal(current_configuration, best_configuration):
            break
        current_configuration = best_configuration
        iteration += 1
        print()

    return best_configuration
```

In contrast to the suboptimal Greedy learner, the **Solver** class is a class that contains methods to deterministically solve the problem of finding the configuration that provides the best expected reward.
Its usefulness lies in the need to have, day by day, a reward value to refer to for the calculation of the regret.

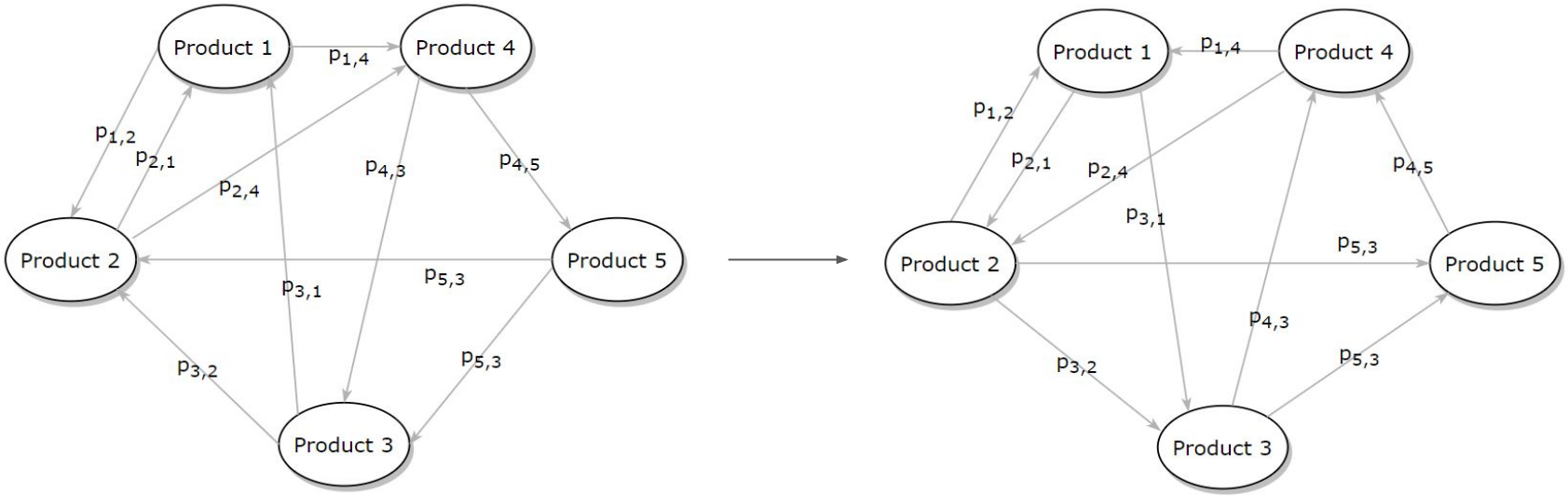
The Solver class, acting as a **clairvoyant** learner, has access to all information about the Environment.
Its operation consists of calculating the total expected reward for each of the 4^5 possible configurations.

For each configuration, the total reward is the sum of the rewards for each product. The reward for each i-th product is calculated as the sum of:

- the reward obtained from the conversions of users who began their visit to the site from page i;
- the rewards obtained from the conversions of users who began their visit to the site from another page but arrived at page i via navigation.

```
class Solver:

    def find_optimal(self):
        arms_shape = (self.n_arms,) * self.n_products
        expected_reward_per_configuration = np.zeros(arms_shape)

        for configuration, _ in np.ndenumerate(expected_reward_per_configuration):
            rewards = np.zeros(self.n_products)
            for start in range(self.n_products):
                common_term =
                    self.conversion_rates[start, configuration[start]] *
                    self.prices[start, configuration[start]] *
                    self.avg_products_sold[start, configuration[start]]
                rewards[start] = common_term *
                    (self.expected_alpha_ratios[start] +
                    self.compute_children_contribute([start], configuration))
            expected_reward_per_configuration[configuration] = np.sum(rewards)

        optimal_configuration = np.unravel_index(
            np.argmax(expected_reward_per_configuration),
            expected_reward_per_configuration.shape)
        optimal_reward = np.max(expected_reward_per_configuration)

        return optimal_configuration, optimal_reward
```

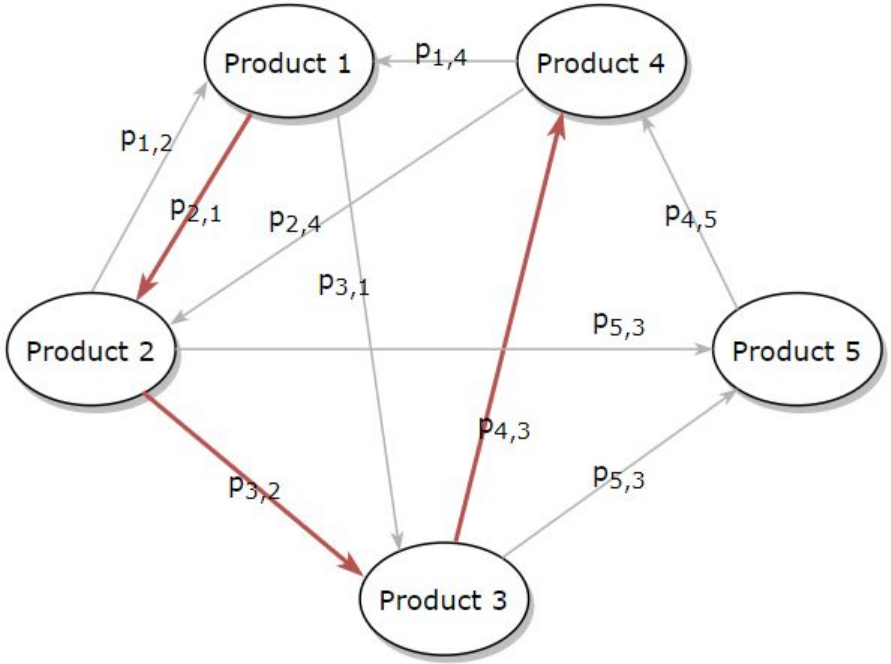
More specifically, an inverse of the product graph is initially created, which will then be used in the calculation of the reward of product i to evaluate each possible **acyclic** path from product j to product i of the original graph (for each $j \neq i$), which, in the inverse graph, corresponds to a path from product i to product j. The contribution from product j can then be weighted according to the probability of following that path.



Solver - example

Let's suppose we want to calculate the reward associated with **Product 1**.

Among all the possible paths in the graph starting from such product we consider the one in red.
By keeping in mind that the one which we are making use of is the **inverse** of the original product graph, this means that we are considering the case in which an user lands in the site through Product 4's page and opens subsequently pages 3, 2 and finally 1.



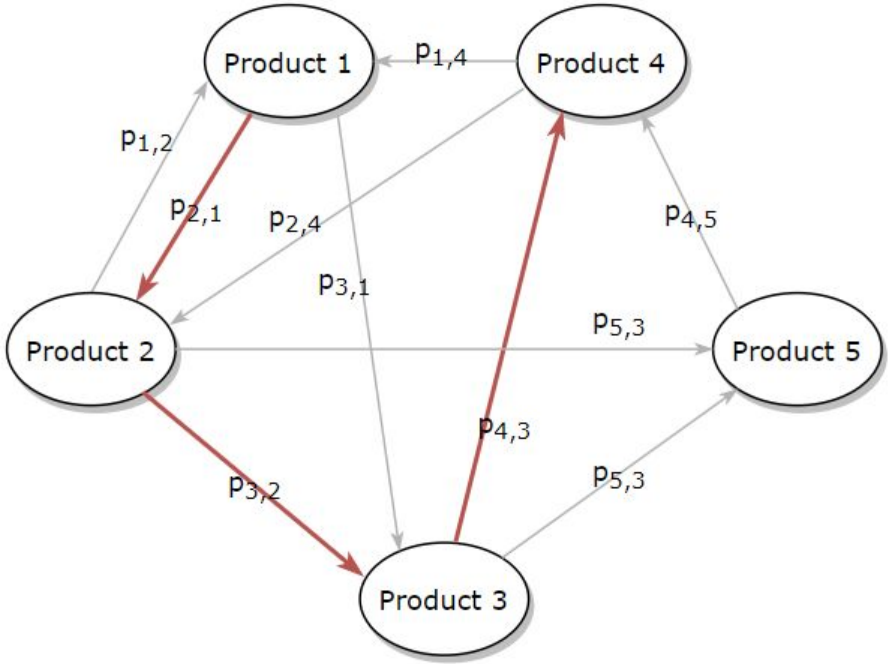
Solver - example

By avoiding to explicit the dependencies from the current configuration, in order to keep the notation lighter, we can call:

- α_i : (expected) alpha ratio of product i
- cr_i : conversion rate of product i
- pr_i : price of product i (function of current configuration)
- ps_i : (average) products sold of product i (function of current configuration)

This means that the **contribution** given by such path is:

$$contribution_{4321} = (\alpha_4\, cr_4\, p_{4,3}) * (cr_3\, p_{3,2}) * (cr_2\, p_{2,1}) * (cr_1\, pr_1\, ps_1)$$

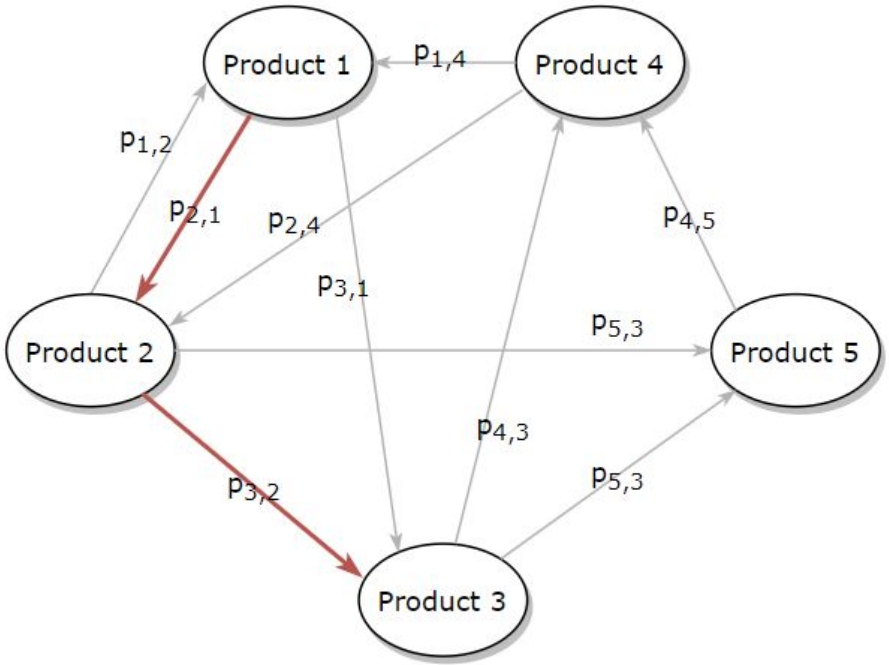


Solver - example

In the same way also **part** of the previously considered path contribute to the reward of Product 1, with a value given by the formula below.

In this case we consider the case in which an user starts his/her visit from **Product 3**:

$$contribution_{321} = (\alpha_3\, cr_3\, p_{3,2}) * (cr_2\, p_{2,1}) * (cr_1\, pr_1\, ps_1)$$



Solver - example

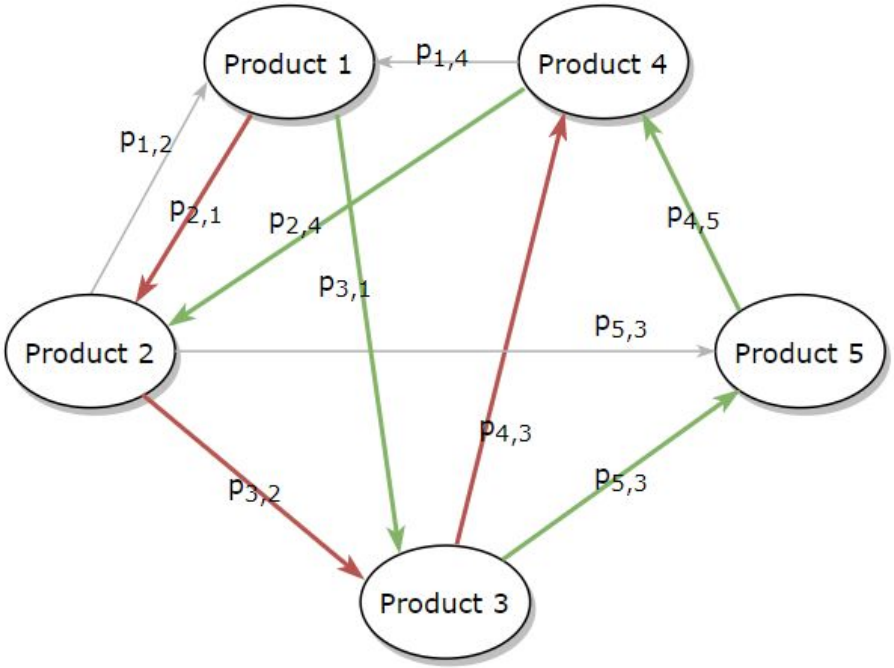
Of course we also need to consider the reward given by the user that started right from Product 1.

In this case the value is simply given by:

$$contribution_1 = \alpha_1 \, cr_1 \, pr_1 \, ps_1$$

The final expected reward value will be given by the sum of the values of all the possible paths, that is:

$$\begin{aligned} reward_1 = \; & contribution_1 + \\ & contribution_{21} + \\ & contribution_{321} + \\ & contribution_{4321} + \\ & contribution_{31} + \\ & contribution_{531} + \\ & contribution_{4531} + \\ & contribution_{24531} + \\ & \ldots \end{aligned}$$



Correctness of the regret

Various tests have shown that Solver contains methods that can successfully find an **optimal configuration**; however, there are certain aspects concerning the calculation of the daily regret that must be taken into consideration.

Each round (or day) is characterised by a high number of operations, the outcome of which is **aleatory** (number of users, drawing of features, success or failure of the purchase, etc.). Therefore, as one can easily imagine, the final reward is strongly bounded to the outcome of these events.

Since the optimal configuration is built on the basis of the best expected reward value, it may happen that in some rounds a non-optimal configuration provides a better reward than the optimal one.
Keeping this approach was preferred instead of using a single optimal ceiling for each round's reward in order to better highlight the properties of the Environment in the final results.

```
class Environment:

    def round(self, pulled_arms, seed=0):
        s = seed
        if seed == 0:
            s = np.random.randint(1, 2**30)
        np.random.seed(s)
        [...]
```

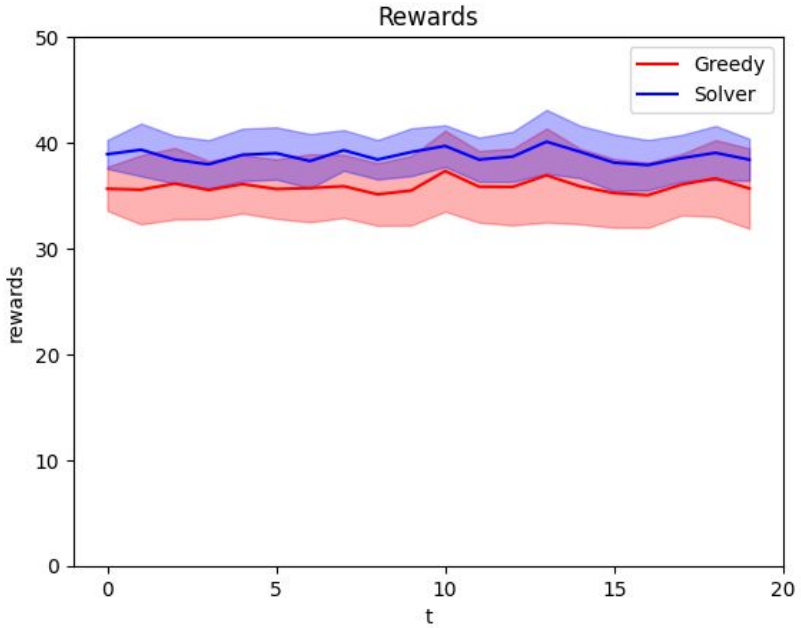
However, in order to minimise the variance of these results, a **seed** for random operations is set at the beginning of each round, thus allowing repeatable events to be generated and making any comparisons more accurate.

Comparison

We can compare the results obtained from playing the Greedy algorithm's optimal configurations to the one found by our Solver. Because the Greedy solver is non-deterministic in nature, the algorithm was run 20 times and its rewards were compared to the Solver's over 20 days.

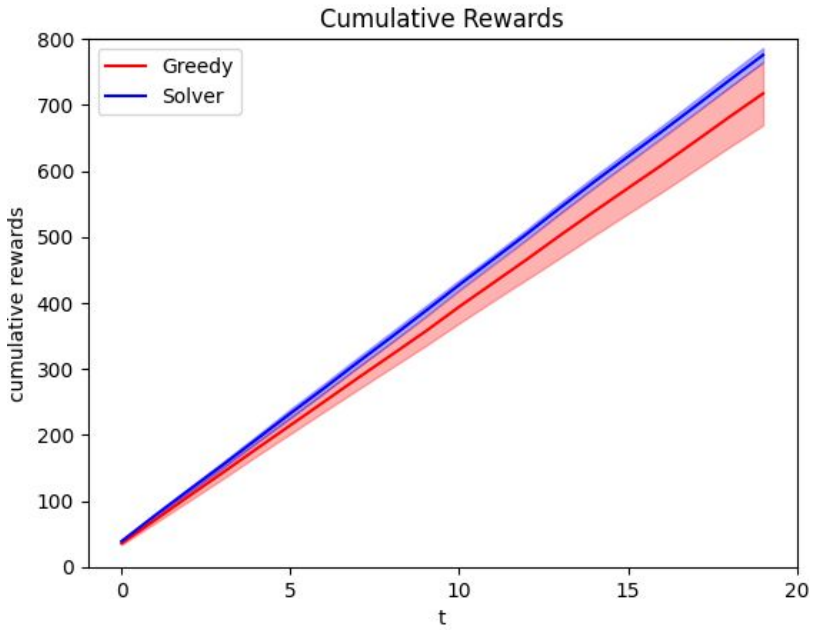
As said before, the **aleatory** nature of the Environment makes it so that the optimal configuration found by the Solver will return a different reward at each round, which might sometimes even be lower than the one obtained from other configurations in specific scenarios.
However, the average behaviors of the two learners clearly show that the **Solver** solution consistently performs better than the Greedy one.

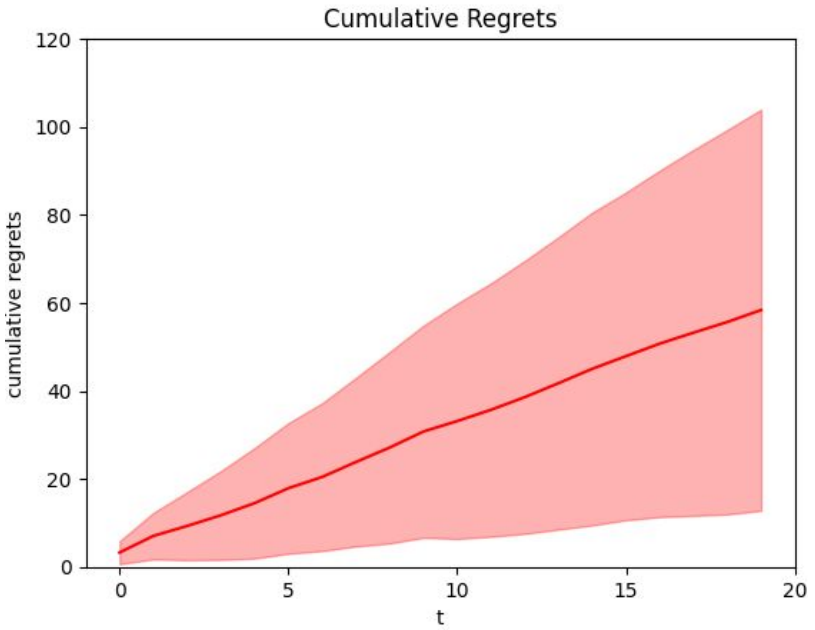
As expected the Solver shows deviations from the norm across its runs as well, which are only being caused by the fluctuations in reward of the Environment.



The cumulative rewards and regrets between the two solutions highlight their differences even better, with the Greedy solution showing a much higher standard deviation because of its intrinsic randomness.

The optimal solution found by the Solver will be used as the **clairvoyant** point of reference for the following algorithms' performances as well.





POLITECNICO
MILANO 1863

Step 3

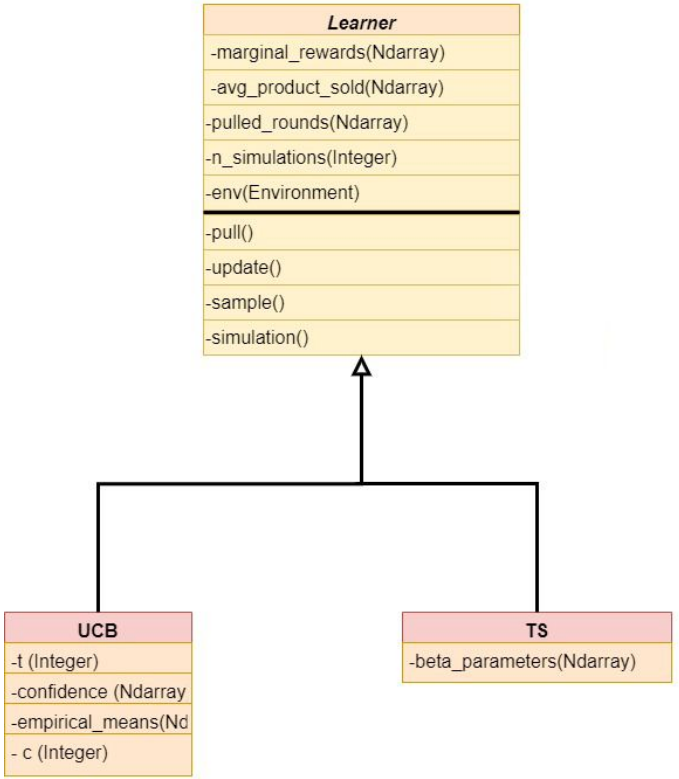
Optimization with uncertain conversion rates

Optimization with uncertain conversion rates

The optimization problem is solved using bandit algorithms implemented by the new Learner class.
Both **UCB** and **Thompson Sampling** approaches are used in order to estimate the missing conversion rates and return the **best performing configuration** of prices for each product.

In the Learner class each product is associated with a different set of **arms**, with a different arm for every possible price point. The bandit algorithms by themselves use these arms to simply estimate a different distribution of conversion rates for each product, independently from the others.
In order to take into account the interdependencies between the products, the estimates returned by the bandit algorithm are adjusted to give more value to price points that enable further chains of purchases.

The prices with the highest resulting expected rewards are played at each round, and the data collected from the Environment is used to update the internal distributions of the bandit algorithms.



Learner class

During each round, the Learner estimates the **reward** associated with the choice of each product-price pair as:

$$\begin{aligned} &\textit{reward (product, price)} = \alpha\ \textit{ratio (product)} * (\\ &\qquad \textit{estimated conversion rate (product, price)} * \\ &\qquad \textit{price} * \\ &\qquad \textit{average units sold (product, price)} + \\ &\qquad \textit{marginal reward (product, price))} \end{aligned}$$

The Learner will choose to play the configuration that maximizes this reward for each product.

Note that for this and all following steps (other than Step 7), a user's features cannot be observed and the data available to the Learner is **aggregated**.
The Learner will then have to estimate an average reward across all user types, in order to return a single configuration that works best for all users. To do this, a **weighted average** is performed on all parameters of the probability distributions of each user class against their class's probability of occurrence (which can easily be extrapolated from the feature probabilities), in order to obtain a single set of parameters used for all calculations.

```python
class Learner:

    def pull(self):
        exp_conversion_rates = self.sample()
        alpha_ratios = np.array(
            [self.get_expected_alpha_ratios()[:self.n_products]] *
            self.n_arms).transpose()
        exp_rewards = alpha_ratios *
            (exp_conversion_rates * self.prices * self.avg_products_sold +
            self.marginal_rewards)
        configuration = np.argmax(exp_rewards, axis=1)
        self.pulled_rounds[np.arange(self.n_products), configuration] += 1
        return configuration
```

The two key values that are not directly taken from the Environment and that need to be evaluated at each iteration are:

- the **estimated conversion rate**, which is sampled from the arms of the bandit algorithm;
- the **marginal reward**, and additional term that takes into account how buying certain products can help make other sales and increase the overall final reward.

Conversion rates - UCB

In the **UCB** algorithm, the **conversion rates** are estimated using the upper confidence bounds of each arm.

At the end of the round, the empirical means of the conversion rates for the chosen configuration are updated using the collected data, while their related confidence becomes:

$$c_t \leftarrow C\sqrt{\frac{\log t}{n}}$$

C = constant
t = number of rounds
n = number of collected samples

```
class UCB(Learner):

    def __init__(self, env: Environment):
        super().__init__(env)
        self.t = 0
        self.confidence = np.ones((self.n_products, self.n_arms)) * np.inf
        self.empirical_means = np.zeros((self.n_products, self.n_arms))
        self.c = 0.2

    def update(self, results: RoundData):
        self.t += 1

        configuration = results.configuration
        conversion_rates = results.conversions / results.visits
        idxs = np.arange(self.n_products)
        n_pulls = self.pulled_rounds[idxs, configuration]

        self.empirical_means[idxs, configuration] =
            (self.empirical_means[idxs, configuration] * (n_pulls - 1) +
             conversion_rates) / n_pulls
        for prod in range(self.n_products):
            for arm in range(self.n_arms):
                n = self.pulled_rounds[prod, arm]
                self.confidence[prod, arm] =
                    self.c * (np.log(self.t) / n) ** 0.5 if n > 0 else np.inf
        self.update_estimates(configuration, results)

    def sample(self):
        return self.empirical_means + self.confidence

    def get_means(self):
        return self.empirical_means
```

Conversion rates - Thompson Sampling

In the **Thompson Sampling** algorithm, the **conversion rates** are modelled by beta distributions, which are sampled to return a current estimate.

After each round, the parameters of the beta distributions are updated using the number of successes (i.e. conversions) and failures (i.e. visits without purchases) obtained in the collected data.

```
class TS(Learner):

    def __init__(self, env: Environment):
        super().__init__(env)
        self.beta_parameters = np.ones((self.n_products, self.n_arms, 2))

    def update(self, results: RoundData):
        configuration = results.configuration
        idxs = np.arange(self.n_products)
        self.beta_parameters[idxs, configuration, 0] += results.conversions
        self.beta_parameters[idxs, configuration, 1] +=
            results.visits - results.conversions
        self.update_estimates(configuration, results)

    def sample(self):
        return np.random.beta(
            self.beta_parameters[:, :, 0],
            self.beta_parameters[:, :, 1])

    def get_means(self):
        return self.beta_parameters[:, :, 0] /
            (self.beta_parameters[:, :, 0] + self.beta_parameters[:, :, 1])
```

Marginal reward

As for the **marginal reward**, this quantity represents the potential rewards that come from the other recommended products after a successful sale of the current product.

In practice, the marginal reward for product A given by another product B is estimated as:

marginal reward (A, B) =

estimated conversion rate (A, priceA) * (1)
reaching probability (A, B) * (2)
estimated reward (B, priceB) * (3)
priceB *
average units sold (B, priceB)

This directly models the scenario where:

(1) the user makes a sale on product A;
(2) the user navigates through the secondary products until they reach product B's page;
(3) the user makes a sale on product B.

```
class Learner:

    def update_marginal_reward(self, configuration):
        reaching_probabilities =
            self.compute_reaching_probabilities(configuration)
        idxs = np.arange(self.n_products)
        for prod in range(self.n_products):
            old_marginal_reward = self.marginal_rewards[prod, configuration[prod]]
            marginal_reward = np.sum(
                self.get_means()[prod, configuration[prod]] *
                reaching_probabilities[prod] *
                self.get_means()[idxs, configuration] *
                self.prices[idxs, configuration] *
                self.avg_products_sold[idxs, configuration]
            )
            n_pulls = self.pulled_rounds[prod, configuration[prod]]
            self.marginal_rewards[prod, configuration[prod]] =
                (old_marginal_reward * (n_pulls - 1) + marginal_reward) / n_pulls
```

Indeed, the **reaching probability** indicates the probability of visiting the page of product B when starting from A, making sales and clicking through the secondary products. This probability needs itself to be estimated from the graph probabilities in the environment.

Step 3
Reaching probabilities

POLITECNICO MILANO 1863

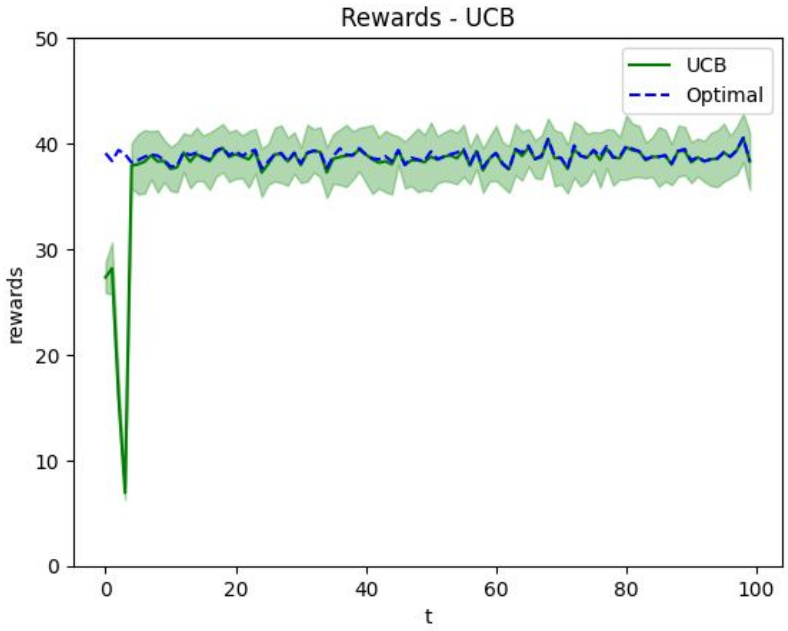
The **reaching probabilities** from a given product are calculated empirically through a series of **simulations** of the environment.
A number of simulated visits is started from the given product in the same way that a real user would buy and click on multiple products, using the graph probabilities known from the Environment and the currently estimated conversion rates to determine when a sale is made.
The total number of times that each product's page is visited is recorded, and then divided by the number of simulations to obtain an estimate of the reaching probability.

This **brute-force** approach is chosen in order to simplify the implementation, which directly mimics the one in the Environment. Moreover, as the resulting edge probabilities tend to be quite low, the graph traversal is fast enough for practical purposes.

```
class Learner:

    def simulation(self, starting_node, configuration):
        visited = np.zeros(self.n_products)
        to_visit = [starting_node]

        while to_visit:
            current_product = to_visit.pop(0)
            visited[current_product] += 1

            buy = np.random.binomial(1,
                self.get_means()[current_product, configuration[current_product]])
            if not buy:
                continue

            secondary_1 = self.secondaries[current_product, 0]
            success_1 = np.random.binomial(1,
                self.graph_probabilities[current_product, secondary_1])
            if success_1 and visited[secondary_1] == 0 and
                secondary_1 not in to_visit:
                to_visit.append(secondary_1)

            secondary_2 = self.secondaries[current_product, 1]
            success_2 = np.random.binomial(1, self.lambda_p *
                self.graph_probabilities[current_product, secondary_2])
            if success_2 and visited[secondary_2] == 0 and
                secondary_2 not in to_visit:
                to_visit.append(secondary_2)

        return visited
```

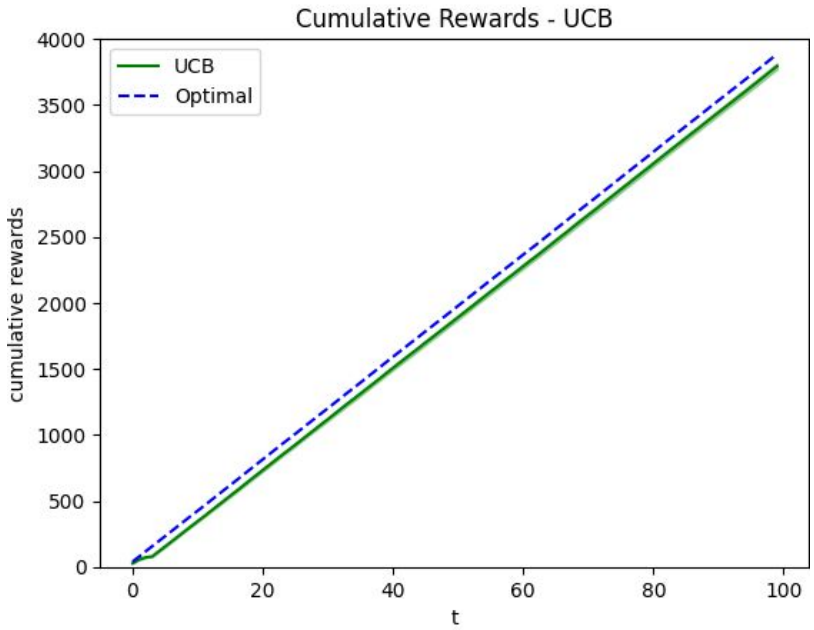
Results

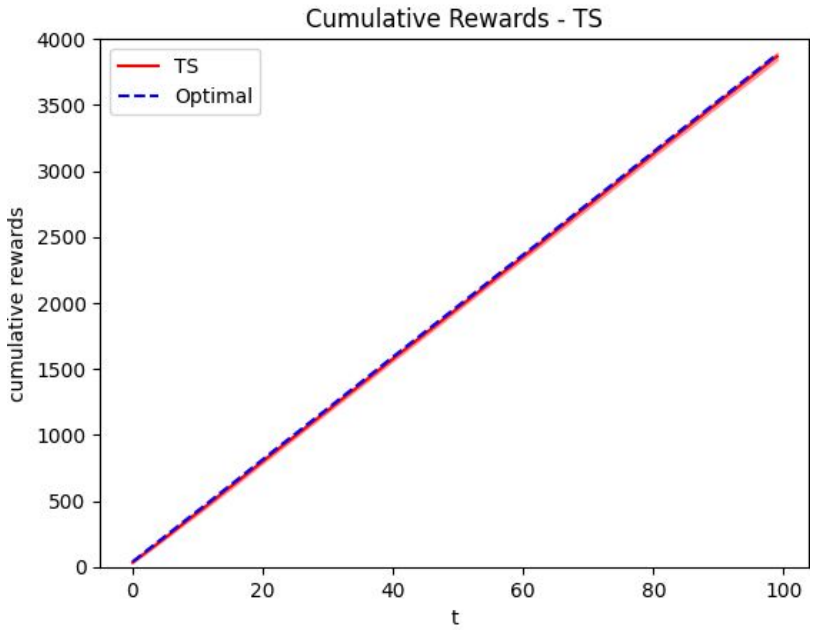
The performance of the Learner class was tested using both UCB and Thompson Sampling bandits for 20 runs of 100 rounds. Here the **average rewards** obtained over all runs are shown together with their standard deviation and compared to the optimal configuration found by the Solver.



Results

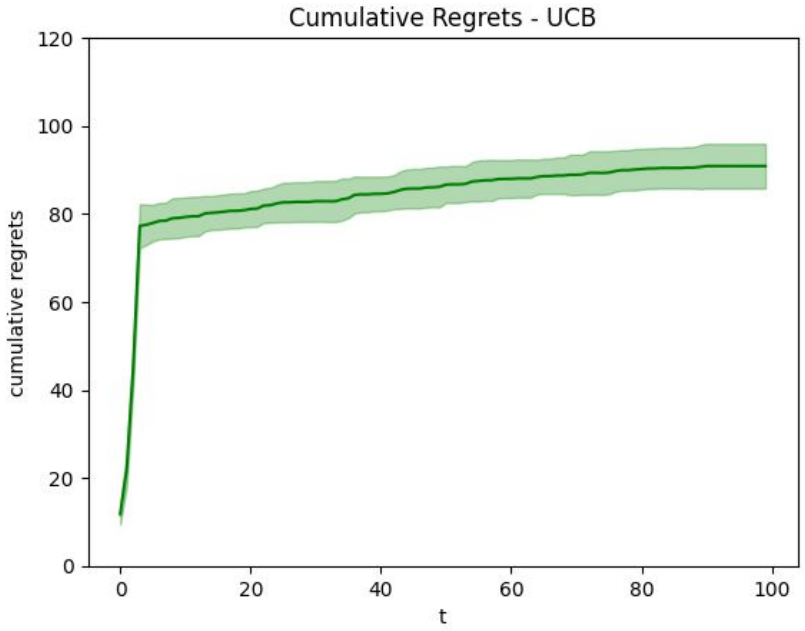
The **Thompson Sampling** version of the algorithm consistently shows better performance, converging faster to the optimal configuration and avoiding deviations from it almost entirely.
The **UCB** approach similarly converges to the optimal solution, but it promotes more exploration even in the later stages of the simulation. UCB also requires that all arms are pulled at least once at the start, giving worse performance in the first few rounds and an overall slightly lower **cumulative reward**.

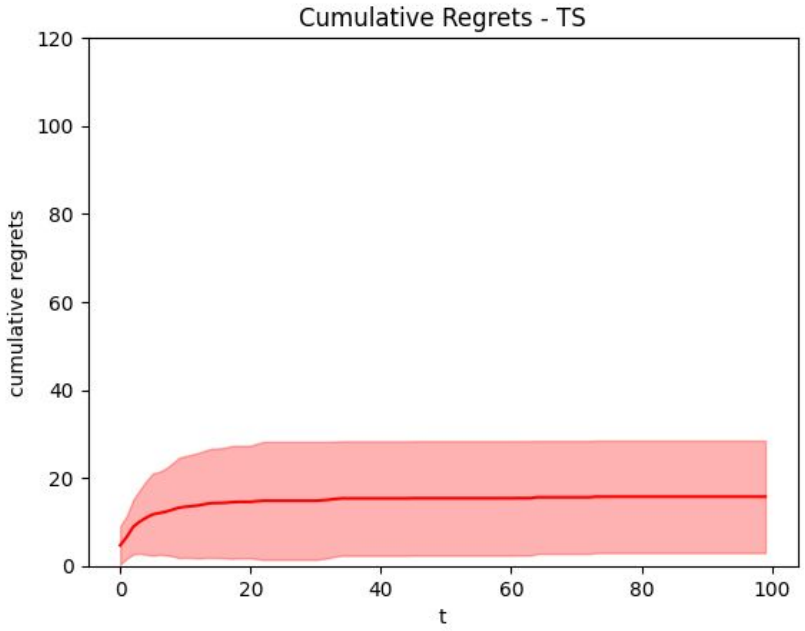




Results

The initial offset in the reward of the UCB algorithm results in a higher starting **cumulative regret**, and its fluctuations lead to a slight continuous growth.
The high **standard deviation** in all distributions (especially notable here with Thompson Sampling) is an inevitable consequence of the high variance in all the random variables of the Environment. Moreover, in the case of the cumulative regrets most of the final regret is given by the performance of the random configuration picked in the first round, when no information is known.





POLITECNICO
MILANO 1863

Step 4

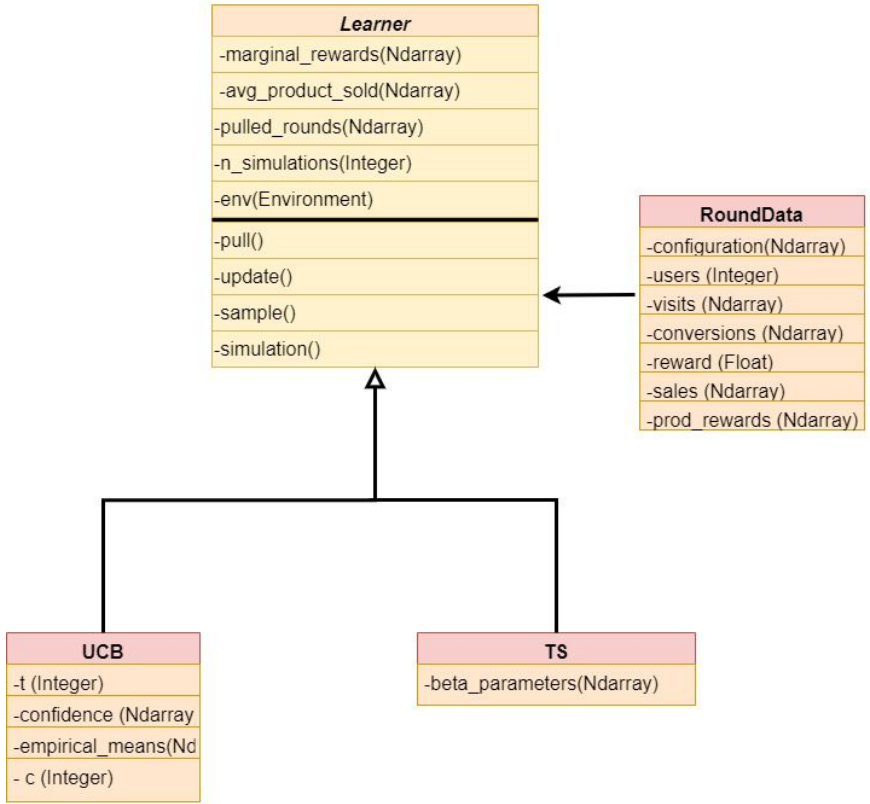
Optimization with uncertain conversion rates, α ratios, and number of items sold per product

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

The Learner class from the previous step is adapted with few modifications to work even when multiple Environment variables are unknown.

No more instances of the bandit algorithms are required, as the ones in charge of estimating the conversion rates still enable the exploration of the state space effectively. Instead, because the actions of each user are perfectly observable by the algorithm, the missing α **ratios** and **average number of items sold per product** are simply estimated by their **empirical mean** taken from the rounds' history.

The **RoundData** class records all the information visible to the Learner during each round simulation, which is used by the Learner to compute the average value of the missing variables. These estimates are used in place of the real Environment probabilities for all calculations, and will converge to said real values with enough data.



α ratios

The α **ratios** represent the initial landing probability on each product's page. Although they are modelled by Beta distribution in the Environment, for simulation purposes we can simply estimate their **empirical mean**, which is updated at each round as:

α *ratio (product)* =
users that first landed on the page (product) /
total number of users

The variable is initialized to give the same weight to each product's page. Note that the ratio α_0 associated with users that land on competitor pages is missing as it is not required for the other calculations.

```
class Learner:

    def __init__(self, env: Environment):

        [...]
        self.alpha_ratios_data =
            np.full((self.n_products + 1, 2), 0)
        self.alpha_ratios_est =
            np.full(self.n_products + 1, 1. / (self.n_products + 1))

    def update_alpha_ratios(self, results: RoundData):
        self.alpha_ratios_data[:self.n_products, 0] += results.first_clicks
        self.alpha_ratios_data[self.n_products, 0] +=
            results.users - np.sum(results.first_clicks)
        self.alpha_ratios_data[:, 1] += results.users
        self.alpha_ratios_est =
            self.alpha_ratios_data[:, 0] / self.alpha_ratios_data[:, 1]
```

Number of items sold per product

The **average number of items sold per product** can be estimated at each round as:

average units sold (product, price) =
total number of purchases (product, price) /
number of conversions (product, price)

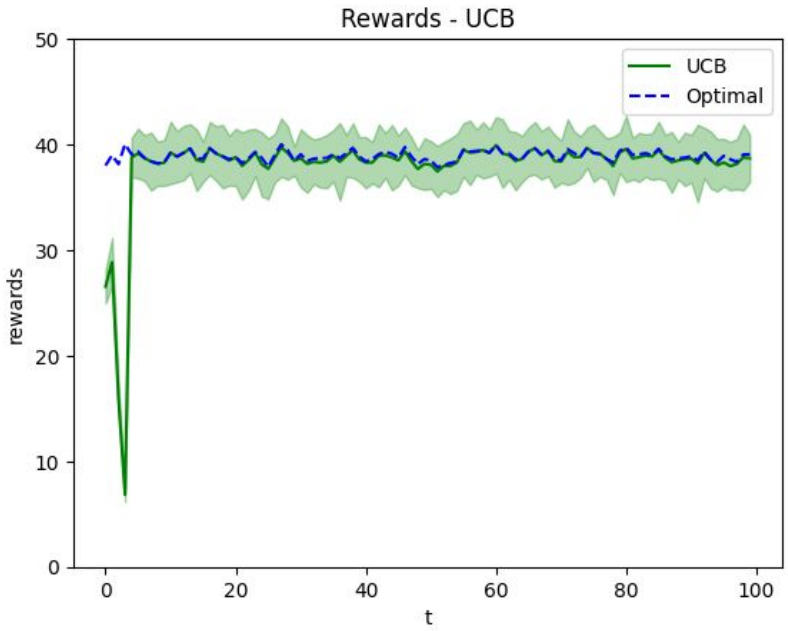
Some attention needs to be given to the **initialization** of this variable. The value should be high enough to promote exploration of unused arms, but at the same time it should stay in line with the real values obtained in the simulations not to throw off any other calculation. A wrong initialization was shown to cause significant downgrades in the performance of the algorithm.

As such, the first round of each run is used to initialize these estimates: the configuration with the lowest prices is played, and all the missing values for the arms that weren't played are set to a value proportional to highest number of items that were sold for any product during that round.

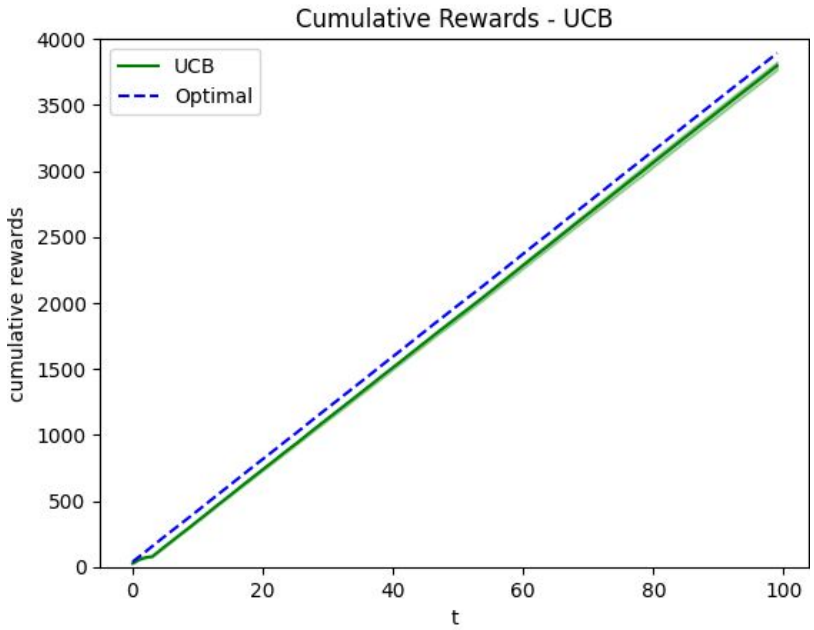
```
class Learner:

    def __init__(self, env: Environment):

        [...]
        self.initialize = True
        self.avg_products_sold_data =
            np.full((self.n_products, self.n_arms, 2), 0)
        self.avg_products_sold_est =
            np.full((self.n_products, self.n_arms), np.inf)

    def update_avg_products_sold(self, configuration, results: RoundData):
        max_found = 0.
        for prod in range(self.n_products):
            sales = self.avg_products_sold_data[prod, configuration[prod], 0]
            conversions =
                self.avg_products_sold_data[prod, configuration[prod], 1]
            sales += results.sales[prod]
            conversions += results.conversions[prod]
            if conversions > 0:
                est = sales / conversions
                self.avg_products_sold_est[prod, configuration[prod]] = est
                if self.initialize:
                    max_found = max(max_found, est)
        if self.initialize:
            self.avg_products_sold_est =
                np.clip(self.avg_products_sold_est, None, 0.8*max_found)
```

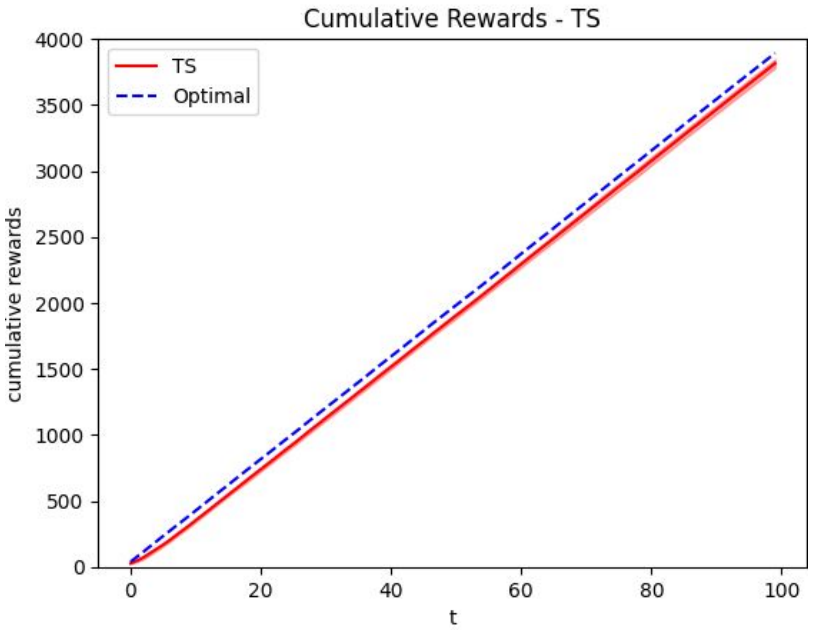
Results

The results from this step are perfectly comparable to the ones from Step 3. Once again, **Thompson Sampling** takes the edge over **UCB** in terms of stability of convergence.



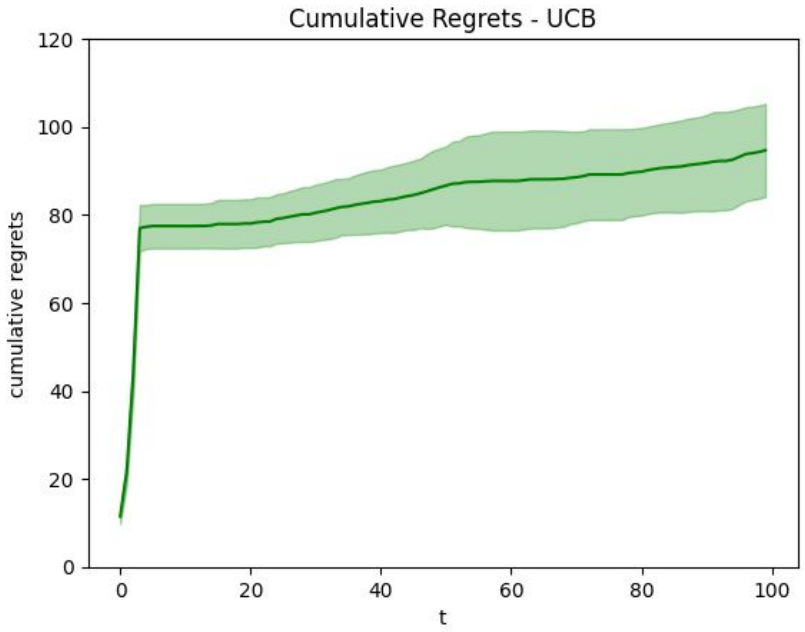
However, both algorithms now need to go through an initialization phase and pull all arms at least once at the start in order to properly initialize the values of the other missing variables. This results in slightly worse initial performance from the two algorithms and lower starting **cumulative reward**, but no change asymptotically.

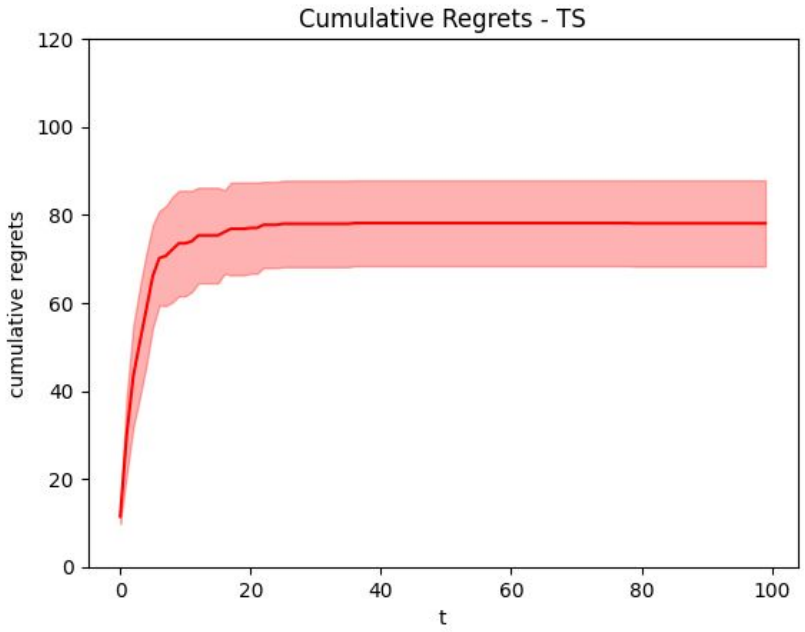




Results

Because both algorithms now take a few rounds to initialize, the convergence speeds of the two algorithms are now approximately the same and the initial jump in **regret** is also similar. However, as expected Thompson Sampling still shows slightly higher stability than UCB at convergence.





POLITECNICO
MILANO 1863

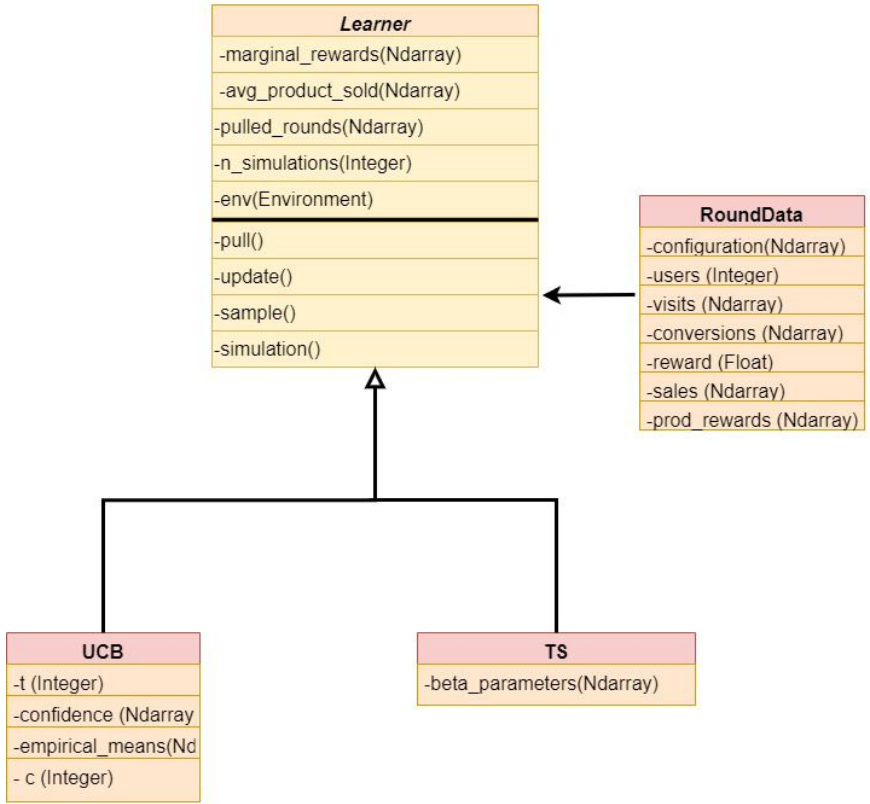
Step 5

Optimization with uncertain graph weights

Optimization with uncertain graph weights

Once again, the introduction of **uncertain graph weights** does not fundamentally alter the Learner implemented for Step 3, as the bandit algorithms still operate simply on the estimation of the conversion rates.

The graph weights necessary for calculations in the Learner can be estimated by their **empirical mean** taken from each round's RoundData, which is updated at each iteration. These estimators will eventually converge to the true probabilities used by the Environment.



Graph weights

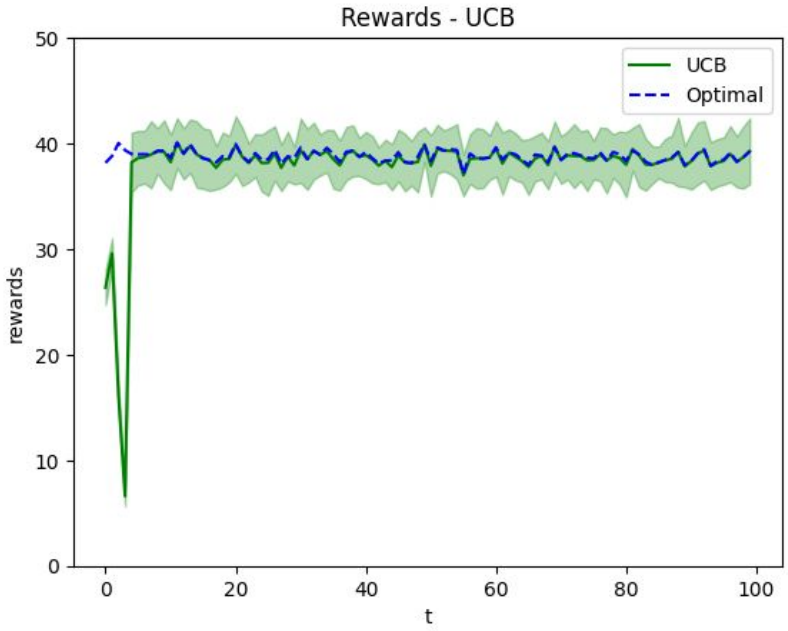
The **graph weight** corresponding to the edge from product A to product B represents the probability of clicking on is updated at each round as:

graph weight (A, B) =
users that click on the secondary product (A, B) /
users that are shown the secondary product (A, B)

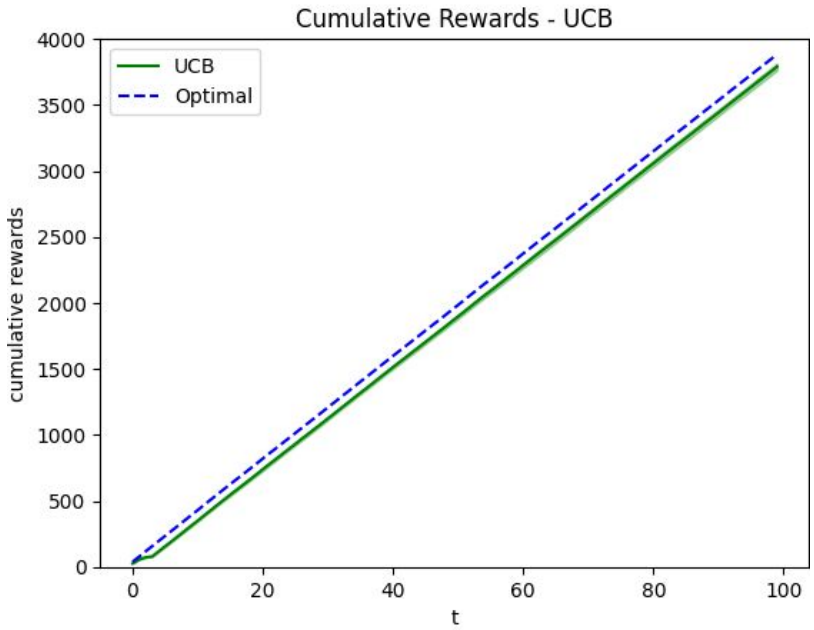
Note that by estimating these values empirically the **lambda** probability associated with secondary products in the second slot is already taken into account. This is not a problem as the secondary products and their order are fixed for each product's page, so the "true" graph weights of products in the second slot are never needed.
The lambda value can then be removed from the internal simulations of the Learner used to estimate the reaching probabilities.

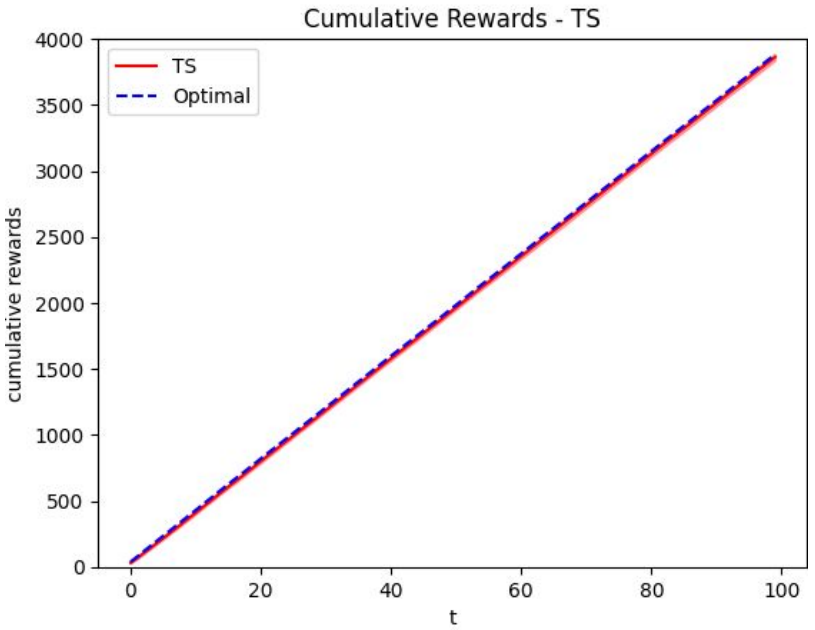
```
class Learner:

    def __init__(self, env: Environment):

        [...]
        self.graph_probabilities_data =
            np.full((self.n_products, self.n_products, 2), 0)
        self.graph_probabilities_est =
            np.full((self.n_products, self.n_products), 1.)

    def update_graph_probabilities(self, results: RoundData):
        self.graph_probabilities_data += results.secondary_visits
        self.graph_probabilities_est = np.divide(
            self.graph_probabilities_data[:, :, 0],
            self.graph_probabilities_data[:, :, 1],
            out=np.zeros_like(self.graph_probabilities_est),
            where=self.graph_probabilities_data[:, :, 1] != 0)
```

The performance of the new algorithm is almost identical to the one of Step 3's Learner.

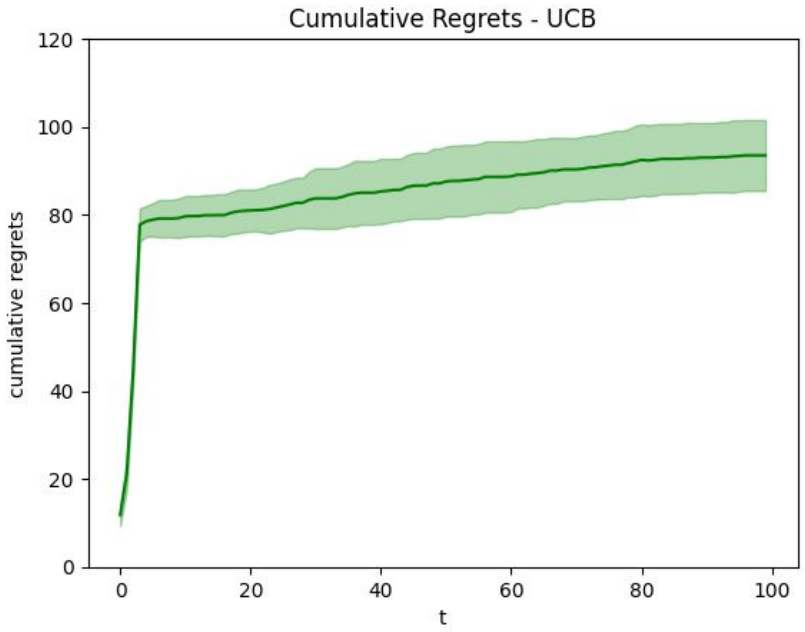


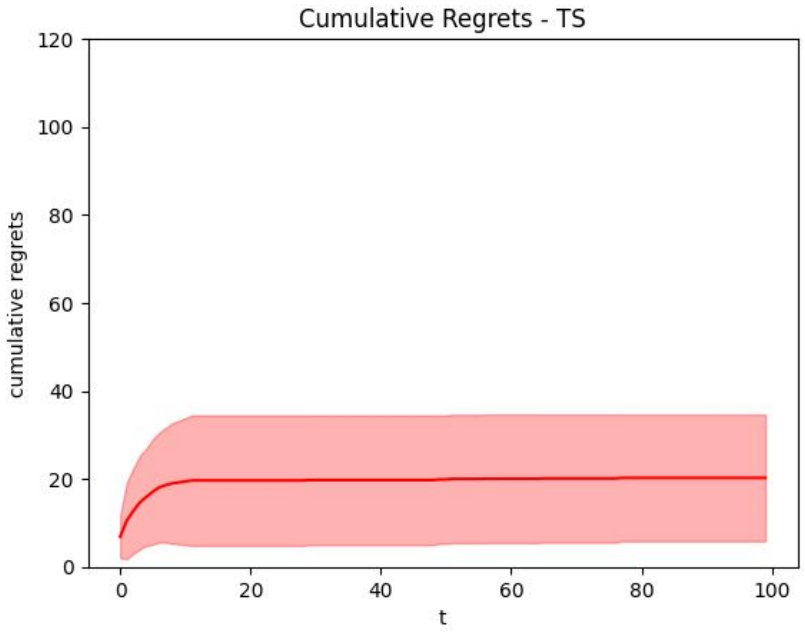
This time, the **initialization** of the graph weights is not tied to the specific configuration chosen by the bandit algorithm, so no rounds are wasted initializing the missing variables.





This allows the **Thompson Sampling** version to achieve the same high performance and fast convergence speed as before, and does not affect the learning of the **UCB** algorithm significantly either.





POLITECNICO
MILANO 1863

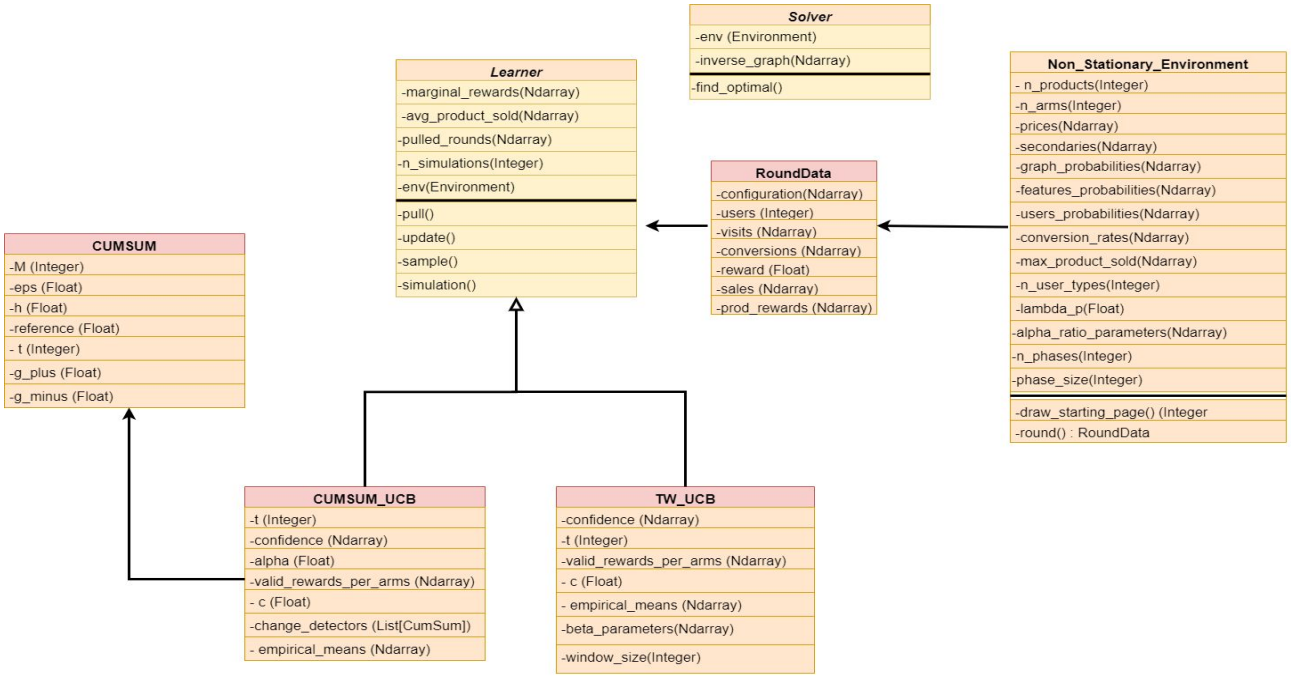
Step 6

Optimization with non-stationary demand curve

Optimization with non-stationary demand curve

We now have a non-stationary demand curve subjected to **abrupt changes**. In this section the Environment class is replaced with the **NonStationaryEnvironment** where the conversion rates of the specific round depend on the different temporal phases of the curve (in our experiments the phases are 4).

Two different approaches have been developed, both inheriting the Learner class implemented in the previous steps: the UCB with change detector that makes use of the **CUMSUM** class whose scope is to alert the algorithm whenever a change in the demand curve is detected, and the UCB with **sliding window**.



UCB with change detector

The **change detection** algorithm is based on the CUMSUM class, which aims to detect abrupt changes in the demand curves by keeping track of their cumulative deviation from the algorithm's estimate.

For the first M rounds, the CUMSUM class builds a **reference point** for the observed conversion rate of each product-price pair by computing its empirical mean over the rounds' data.
After that, if a round's estimated conversion rate exceeds the reference point by more than $\pm\,\varepsilon$, this **deviation** is added to a respective positive or negative cumulative deviation counter. An abrupt change in the demand curve is finally registered by the class and notified to the learner when one of these counters hits another threshold h.

With some manual testing the values of $M = 15$, $\varepsilon = 0.05$, $h = 0.15$ were shown to give good performance to the algorithm.

```python
class CUMSUM:

    def __init__(self, M, eps, h):
        self.M = M  # first samples to calculate the reference point
        self.eps = eps  # epsilon is in the formula of the deviation from the
            reference
        self.h = h  # upper and lower bound for an abrupt change
        self.reference = 0  # reference is the empirical mean over the first M
            samples
        self.t = 0
        self.g_plus = 0  # cumulative pos deviation of an arm until time t
        self.g_minus = 0  # cumulative neg deviation of an arm until time t

    def update(self, data: RoundData, arm):
        self.t += 1
        if self.t <= self.M:
            # warm up to compute the reference
            self.reference += (data.conversions[arm] / data.visits[arm]) / self.M
            return 0
        else:
            # Positive and negative deviation from the reference point
            s_plus = ((data.conversions[arm] / data.visits[arm]) -
                self.reference) - self.eps
            s_minus = -((data.conversions[arm] / data.visits[arm]) -
                self.reference) - self.eps
            self.g_plus = max(0, self.g_plus + s_plus)
            self.g_minus = max(0, self.g_minus + s_minus)
            return self.g_plus > self.h or self.g_minus > self.h  # True if a
                change is detected
```

UCB with change detector

The UCB learner associated with this change detection algorithm only differs from the base one for its *update* function. If the CUMSUM class detects a change in the demand curve, all the information collected up to that time (the **marginal rewards**, the **number of rounds** in which a price has been pulled and the **CUMSUM** collected statistics) will be discarded because not compliant with the current phase anymore.

In the case in which no change is detected, the data coming from the next round will be used to estimate the conversion rate and to update the upper confidence bound as usual.

Note that the change detection algorithm operates separately on each of the different products (effectively modelled by 5 separate bandits) in order not to lose any information unless strictly necessary.

```
class CUMSUM_UCB(Learner):

    def update(self, data: RoundData):
        self.t += 1
        pulled_arm = data.configuration

        for prod in range(self.n_products):
            if self.change_detection[prod][pulled_arm[prod]].update(
                data, pulled_arm[prod]):
                # If this is True, it means that there was an abrupt change.
                # Then we finally reset all the CUMSUM parameters
                self.valid_rewards_per_arms[prod][pulled_arm[prod]] = []
                self.pulled_rounds[prod, pulled_arm[prod]] = 0
                self.marginal_rewards[prod, pulled_arm[prod]] = 0
                self.change_detection[prod][pulled_arm[prod]].reset()

            # Update of the other variables in any case and upper-bound
                computation
            self.valid_rewards_per_arms[prod][pulled_arm[prod]].append(
                data.conversions[prod]/data.visits[prod])
            self.empirical_means[prod, pulled_arm[prod]] =
                np.mean(self.valid_rewards_per_arms[prod][pulled_arm[prod]])
            total_valid_samples = np.sum(self.pulled_rounds[prod])
            for a in range(self.n_arms):
                n_samples = self.pulled_rounds[prod][a]
                self.confidence[prod][a] =
                    self.c * (np.log(total_valid_samples)/n_samples) ** 0.5
                    if n_samples > 0 else np.inf
        self.update_marginal_reward(pulled_arm)
```

UCB with sliding-window

The **UCB sliding-window** algorithm works like the standard UCB except for the insertion of a parameter (the **window size**) that at each round ensures that only the most recently observed samples are taken into account. This is used as a baseline to compare the results of our change detector.

At each step the **sliding window** shifts to the right including the new sample and, at the same time, excluding the oldest sample that was available at the previous step. In this way it is possible to deal with **abrupt changes**. In fact, old samples, that might be observed during a previous phase of the demand curve, cannot affect the computation of the confidence bounds in the current phase.

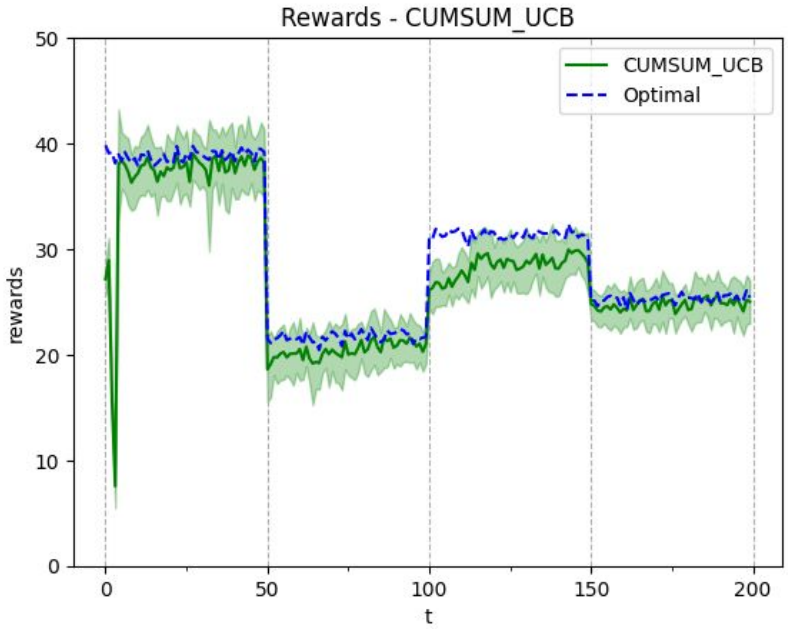
In our experiment the window size was set to 31 after some attempts. This value guarantees a good level of stability for the results and it respects the condition:

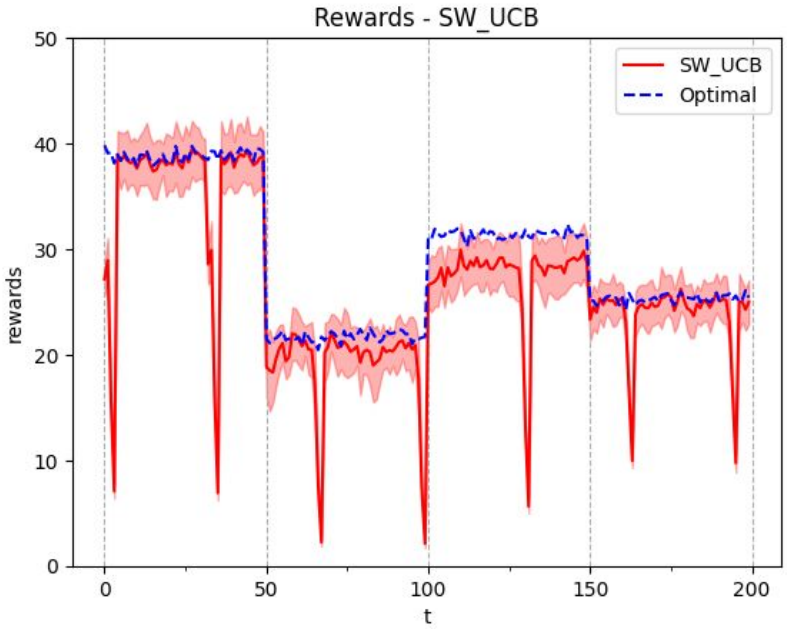
$\tau \approx 2\sqrt{T} \propto \sqrt{T}$ (with horizon $T = 200$ in our case).

```
class SW_UCB(Learner)

    def update(self, results: RoundData):
        self.t += 1
        configuration = results.configuration  # pulled arms during last round
        conversion_rates = results.conversions / results.visits
        self.pulled_arms_timeline.append(configuration)  # to store pulled arms in
            time
        for prod in range(self.n_products):
            for arm in range(self.n_arms):
                self.rewards_per_arms[prod][configuration[prod]].append(
                conversion_rates[prod])  # shape n_prod X n_arms

        # select last window_size pulled configurations
        valid_configurations =
            np.array(self.pulled_arms_timeline[-self.window_size:])
        # np.bincount counts the number of times each arm has been pulled for each
            product
        self.pulled_rounds = np.array(
            [np.bincount(pulls_per_prod, minlength=self.n_arms)
            for pulls_per_prod in valid_configurations.T])

        for prod in range(self.n_products):
            for arm in range(self.n_arms):
                n = self.pulled_rounds[prod][arm]  # to select only valid rewards
                    for update
                self.empirical_means[prod, arm] = np.sum(
                    self.rewards_per_arms[prod][arm][-n:]) / n if n > 0 else 0
                self.confidence[prod, arm] =
                    self.c * (np.log(self.t) / n) ** 0.5 if n > 0 else np.inf
```

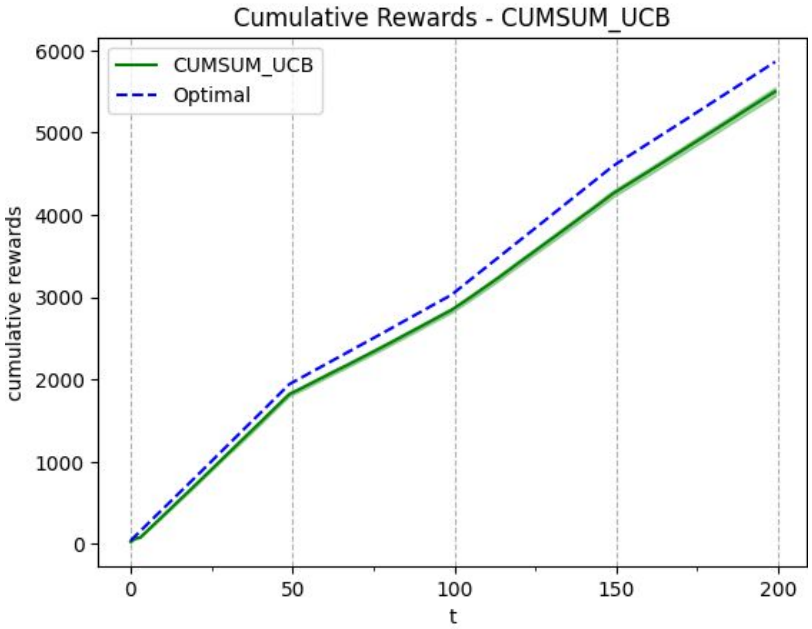
Results

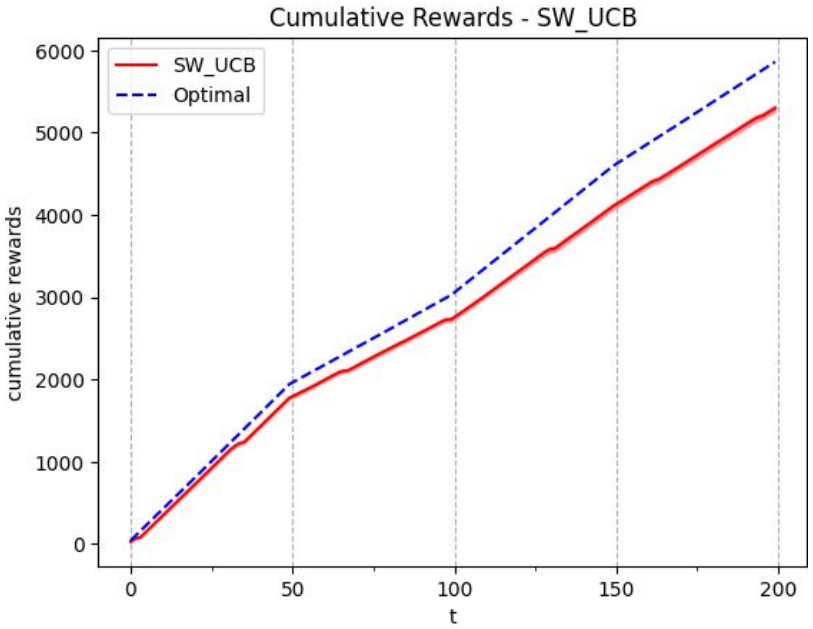
The **reward** plots for the two algorithm show better overall performance from the **change detection** algorithm compared to the sliding-window version. Both algorithms are effective in immediately identifying changes to the demand curve and responding accordingly, converging to similar optimal configurations. However, the sliding-window approach displays **negative spikes** in the reward that are the result of a new tentative of exploration by the UCB algorithm after that the last pull on a given suboptimal arm exits the window. Indeed, this “problem” occurs with a frequency bounded to the value of the window size.





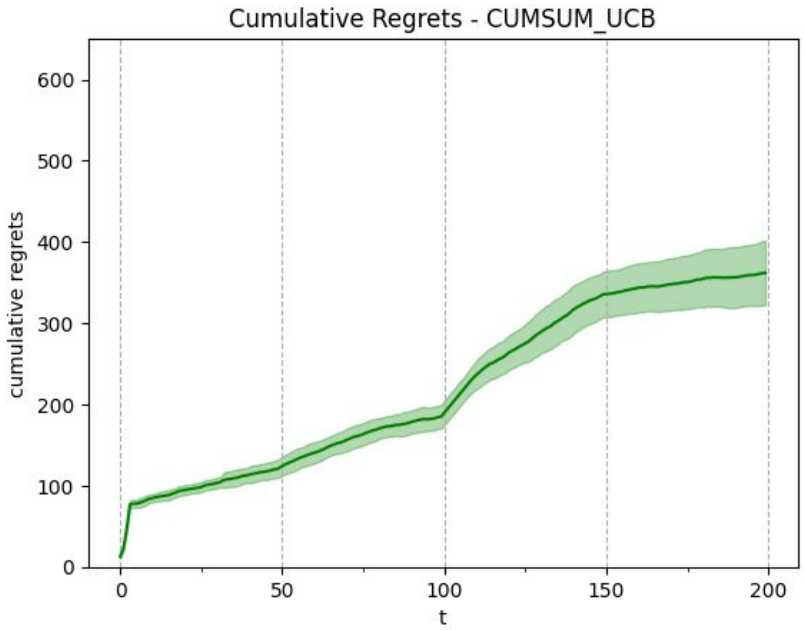
The negative reward spikes result in a worse cumulative reward curve for the sliding-window algorithm, which once again follows a similar trend to the CUMSUM version but meets some stops that lower its final value.

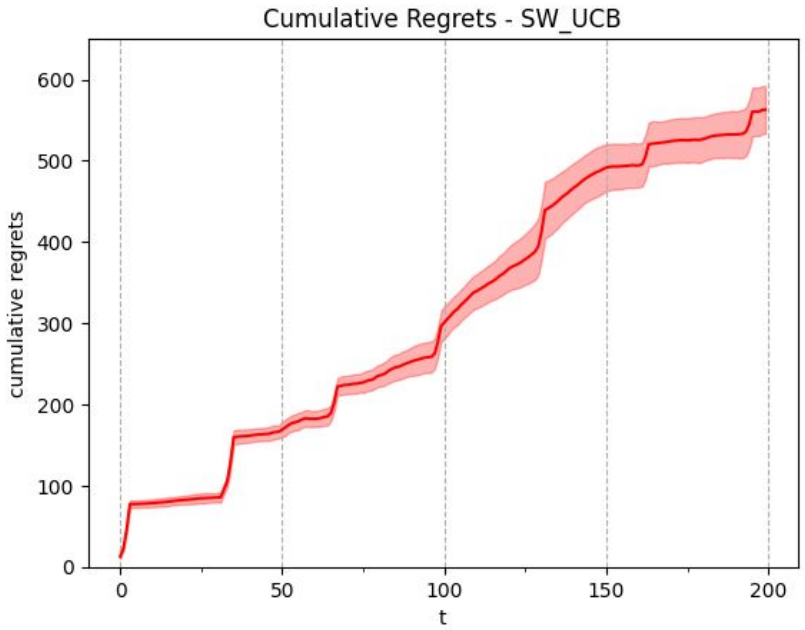




Results

The cumulative regrets curves highlight these trend even better, clearly denoting the steps where the sliding-window algorithm gave the wrong configuration.





POLITECNICO
MILANO 1863

Step 7

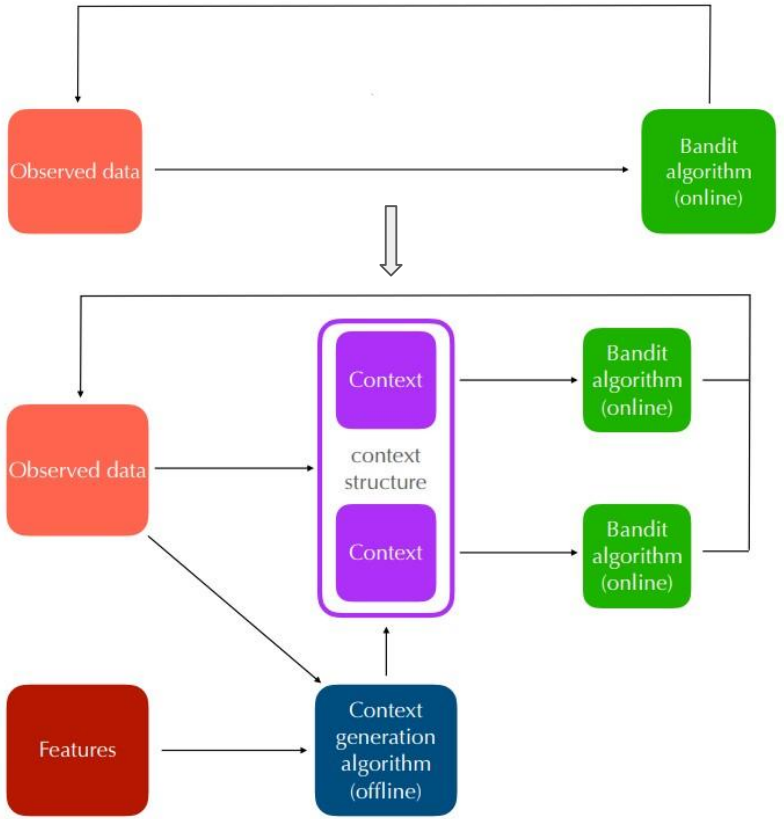
Context generation

Step 7
Context generation

So far no distinction between user classes has been made. The data observed from the environment were treated as if there was only one possible type of customer on the website and the bandit algorithms learnt **aggregate** models.

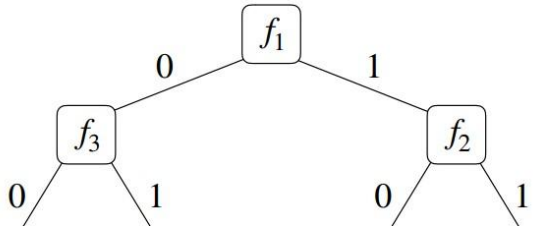
In this step the concept of **context** is introduced. Given that every user is identified with some features, a context represents a possible partition of space of these features. By associating a different learner to each context it is possible to fit a specific model for each class of customers obtaining an increase of the general performances.

As per specification, a context generation algorithm is run every 14 rounds in order to find the best performing partition of features.



Context generation algorithm

The approach of the context generation algorithm provides for creating partitions of the feature space during the execution time and then checking if performing this **split** procedure gives better results than not doing it. If yes, then two new contexts are generated from the starting context.



To evaluate the success of the split, the following comparison is performed:

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

With $p_{c\,k}$ probability that **context** k occurs and $\mu_{a,c\,k}$ the lower bound on the **expected reward** for the learner associated to **context** k. For rewards following Bernoulli distribution, the **Hoeffding Bound** can be used to represent expected reward lower bound (with δ the confidence and Z the set of data):

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

Algorithm implementation

Because each user's class is defined by 2 binary features, there are 4 total combinations of values (class types) associated to only 3 types of user.

The developed algorithm follows a **greedy** approach based on a tree structure starting from the root **ContextNode** class, where all classes of users are aggregated. For 14 rounds (size of the *split_step*) the learners associated to the **context leaves** are trained, then, for all the leaves, for all possible not-already-splitted features, two **children context node** are created.

At first, the learners linked to these new context nodes are initialized by training on the data gathered so far by taking into account only those concerning the respective class types, then evaluation on the splits is performed.

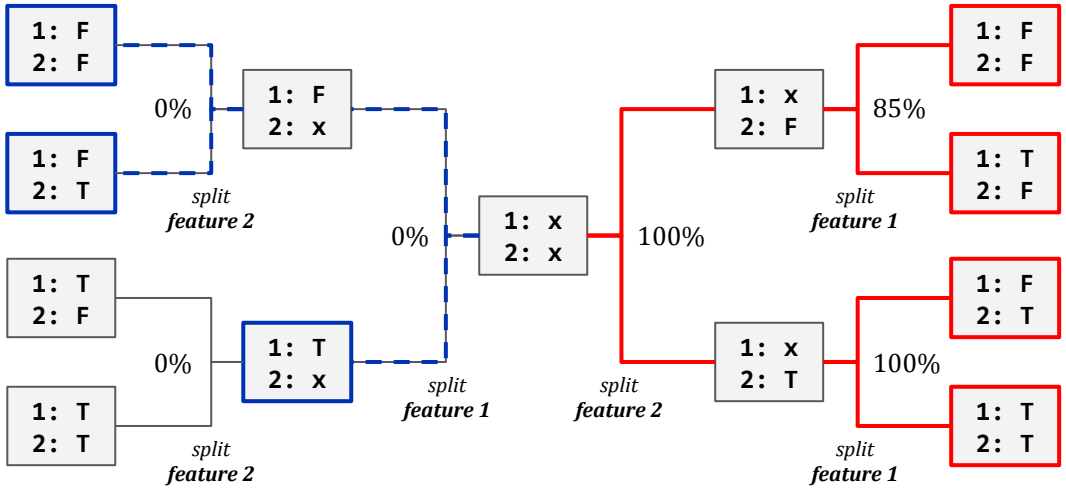
This procedure allows to check as soon as possible if a split of one of the expandable features is proficient, or if it is worth to keep data aggregated as in the parent context. The algorithm stops when no more leaves are splittable or at the end of the execution.

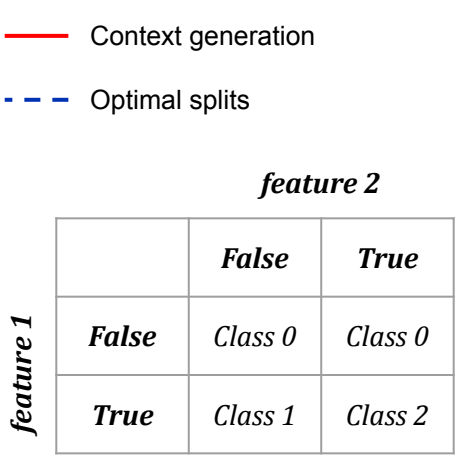
```
def split(self):
    splittable_features = self.get_splittable_features()
    if not splittable_features:
        return False
    parent_x_ = np.sum(np.mean(self.learner.get_means(), axis=1))
    t = RoundsHistory.get_number_rounds(type(self.learner))
    best_lower_bound = parent_x_ - (- np.log(self.delta) / (2 * t)) ** 0.5
    fp = self.feature_probabilities
    for feature in splittable_features:
        if feature == 1:
            left = ContextNode(self.env, type(self.learner),
                feature_1=False, feature_2=self.feature_2, delta=self.delta)
            right = ContextNode(self.env, type(self.learner),
                feature_1=True, feature_2=self.feature_2, delta=self.delta)
        elif feature == 2:
            left = ContextNode(self.env, type(self.learner),
                feature_1=self.feature_1, feature_2=False, delta=self.delta)
            right = ContextNode(self.env, type(self.learner),
                feature_1=self.feature_1, feature_2=True, delta=self.delta)
        left_x_ = np.sum(np.mean(left.learner.get_means(), axis=1))
        left_lower_bound = left_x_ - (- np.log(self.delta) / (2 * t)) ** 0.5
        right_x_ = np.sum(np.mean(right.learner.get_means(), axis=1))
        right_lower_bound = right_x_ - (- np.log(self.delta) / (2 * t)) ** 0.5
        children_lower_bound = (1 - fp[feature - 1]) * left_lower_bound +
            fp[feature - 1] * right_lower_bound
        if children_lower_bound >= best_lower_bound:
            best_lower_bound = children_lower_bound
            self.left = left
            self.right = right
    return self.left is not None and self.right is not None
```

Results - context generation

Both the UCB and Thompson Sampling learners from Step 4 were adapted to work with the new contexts, with a new **RoundsHistory** class being used to save all data from previous rounds for training of new ContextNodes. The Environment variables were also changed to work with different user types.

The graph below shows the results of the context generation algorithm across multiple runs. The greedy, **suboptimal** algorithm fails to identify the correct split of features as it always prefers to make the first split over *feature 2* instead of *feature 1*. Instead, most of the time the algorithm ends up splitting the 4 possible combinations of features into 4 separate classes. This way the Learners are still able to converge to the **optimal configuration** for all users, as the two types of users belonging to Class 0 have the same Environment parameters and always end up with the same final configuration.

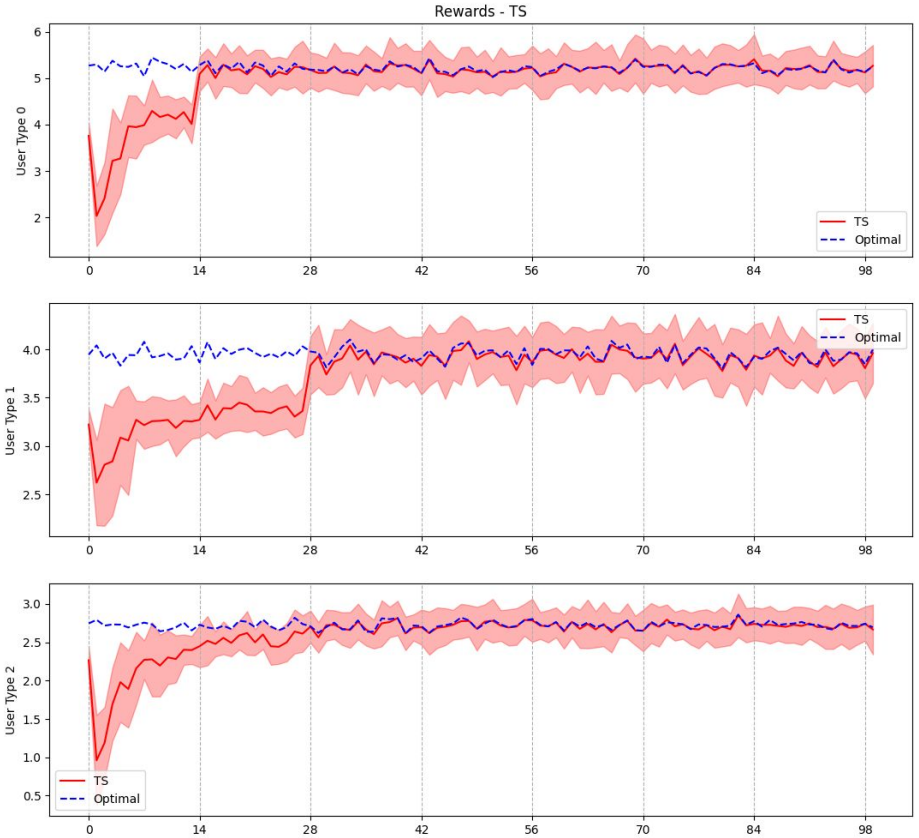




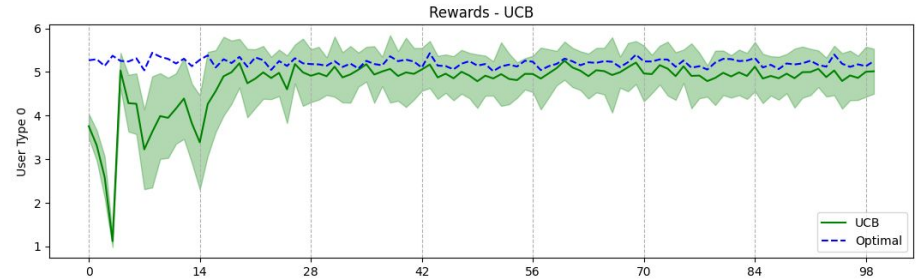
Results

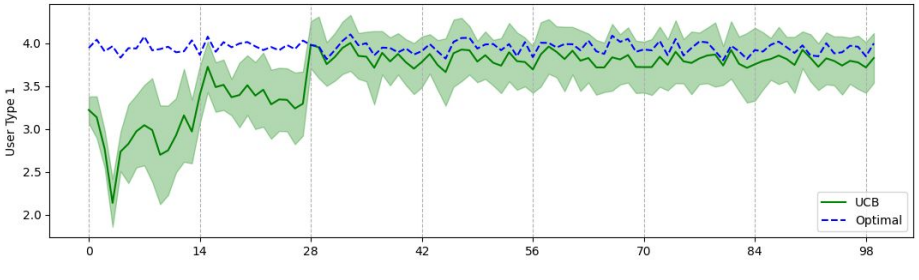
The performance of the two Learners was evaluated against the individual optimal configurations for each user class, found once again by a modified Solver.

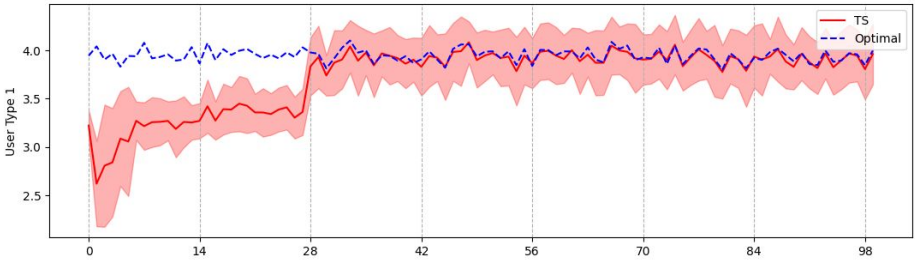
The data shows that both algorithms are able to successfully optimize over the separate classes, with **Thompson Sampling** clearly outperforming **UCB** overall. Each split in the user classes leads to a clear improvement in the total reward, until optimal configurations are found after two splits.
Thompson Sampling's higher convergence speed and more stable configuration lead to significant improvements over UCB both in terms of cumulative reward and cumulative regret.

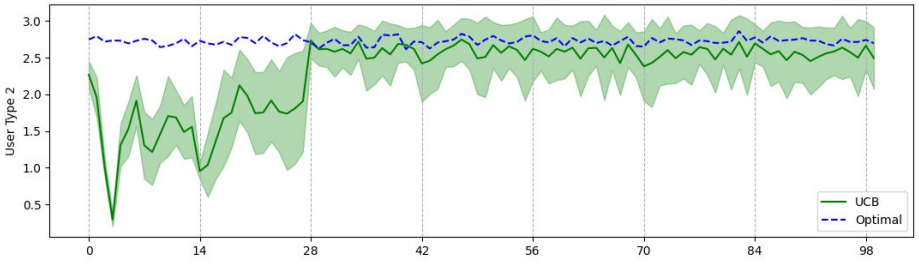


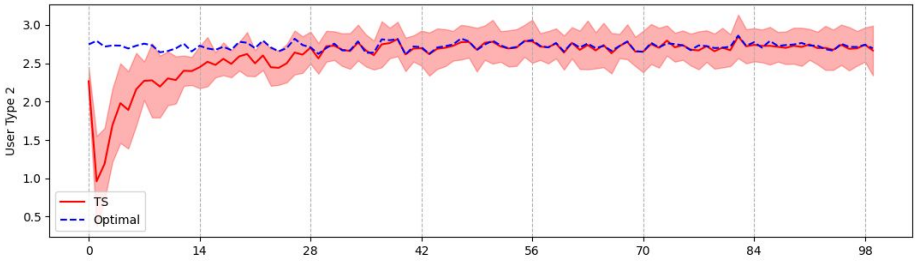
Step 7
Results





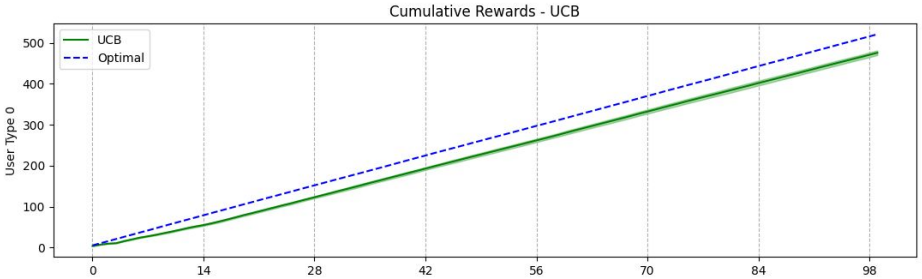


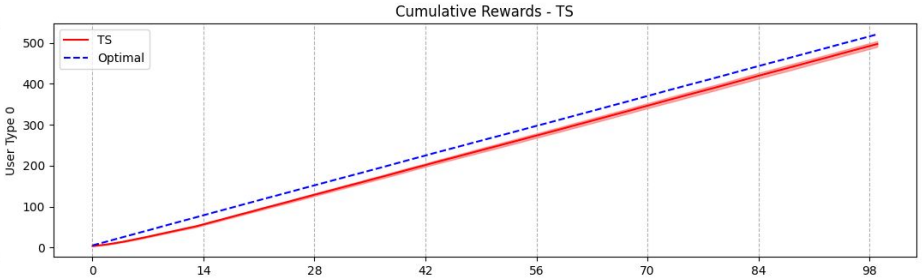


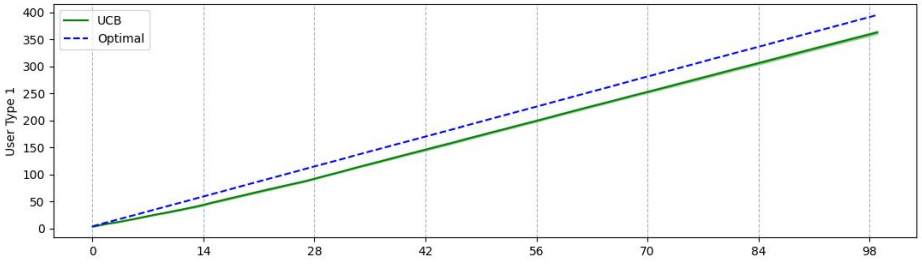


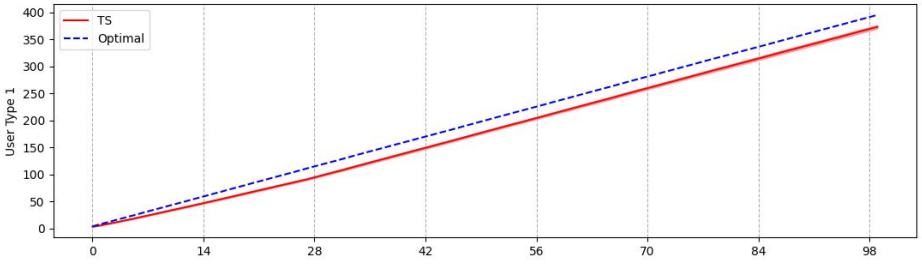
Step 7
Results

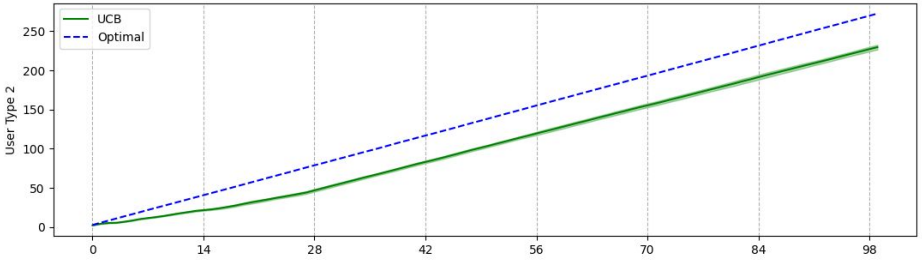
POLITECNICO MILANO 1863

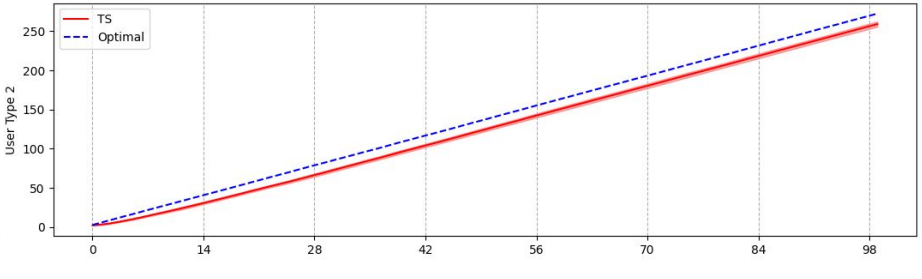






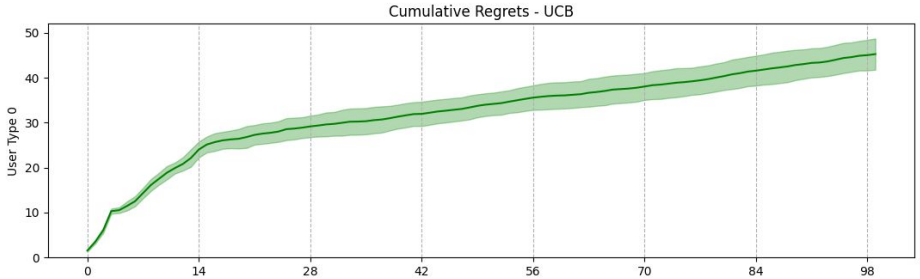


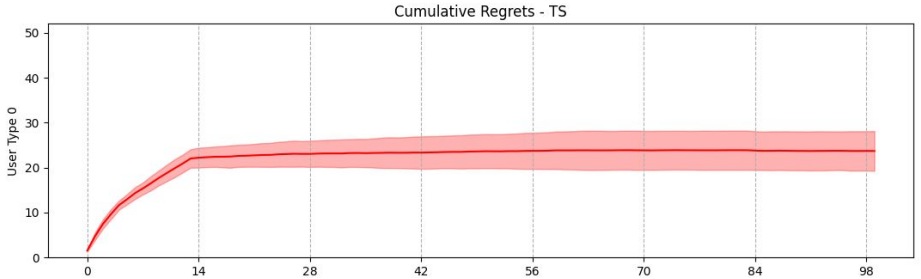


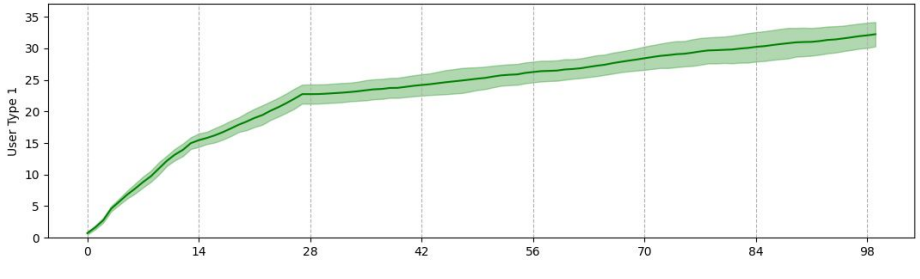


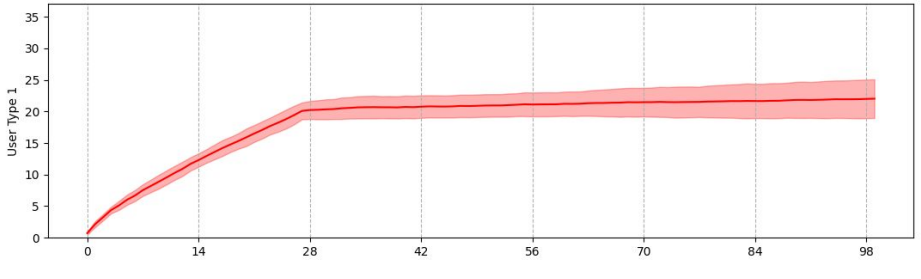
Step 7
Results

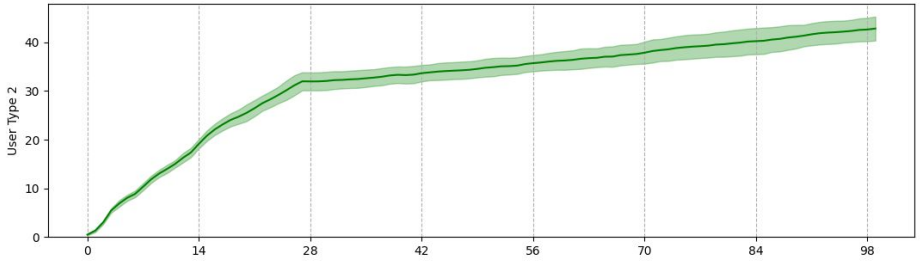
POLITECNICO MILANO 1863

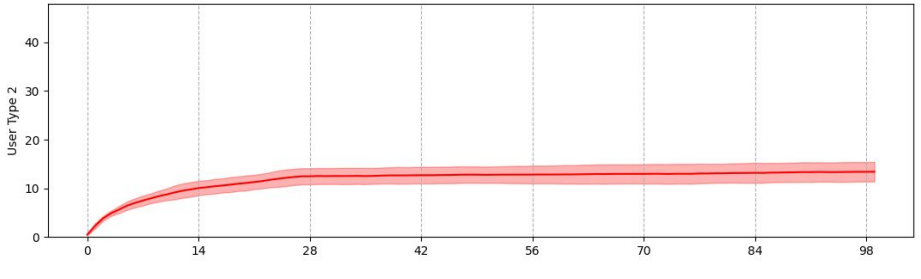












Thank you for your attention!