Presentation of OLA project

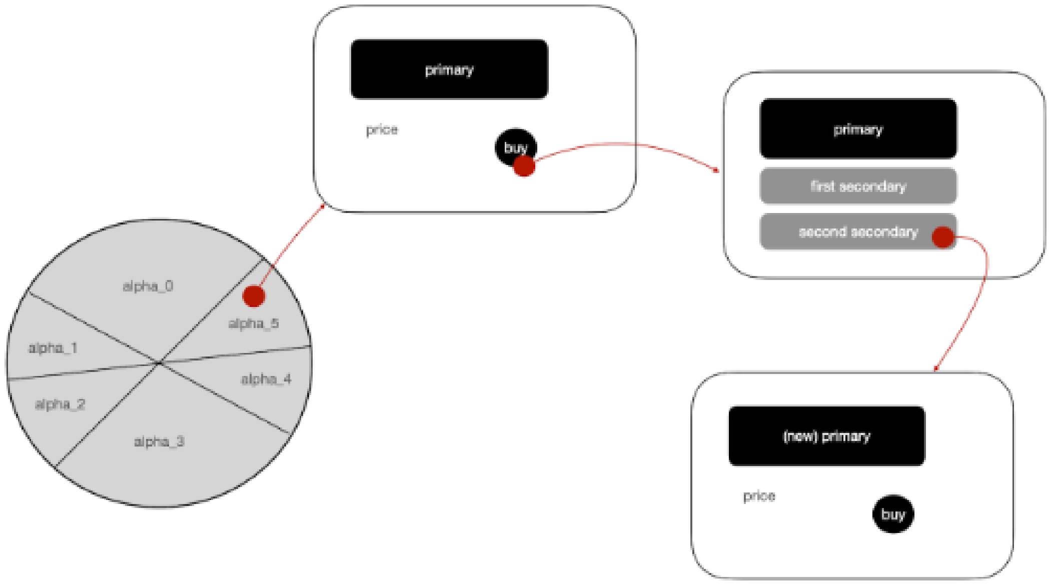
Pricing

2021/2022

Alice Brugnoli, Marco Tonnarelli, Mattia Sabella, Andrea Isgrò, Giovanni Clini

The e-commerce

Every day a random number of customers enters the e-commerce. Every single customer land on a webpage in which one of the 5 products is primary. If this product is added to the cart by the user, then two other products are recommended by the system in different slots. If the user clicks one of them, called secondary, then a new page is loaded in which the clicked one is displayed as primary together with its own price.



STEP 1 - Environment

The actual environment is composed of the following classes:

- Environment
- Non Stationary Environment
- Contextual Environment

There are also other two classes used in the scope of the environment:

- Customer Class
- Context Class

STEP 1 - Environment

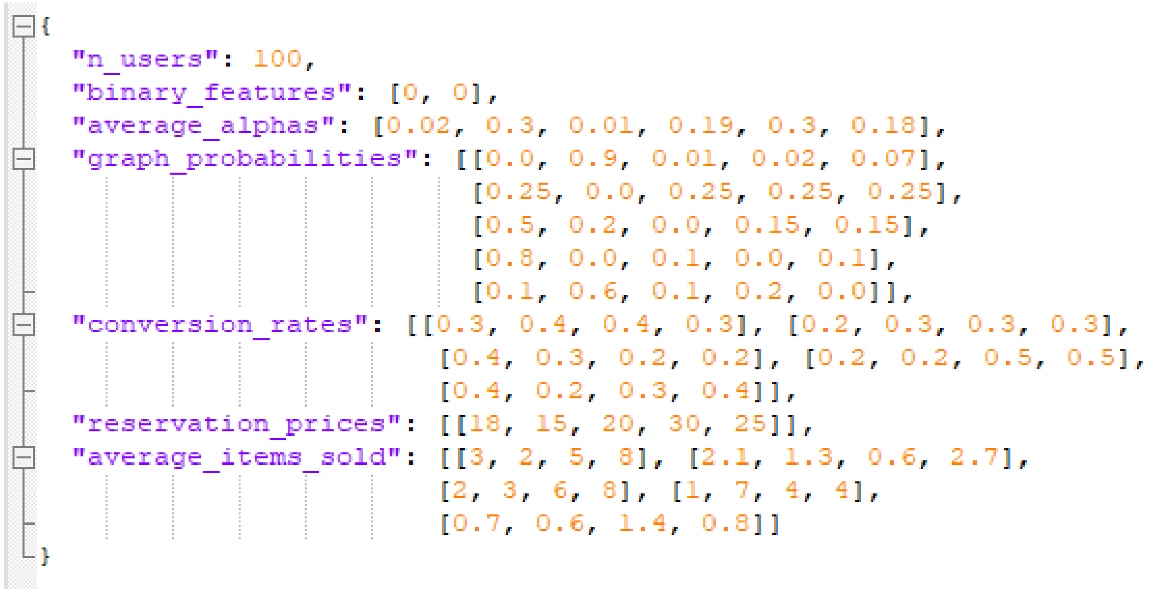
Environment and Contextual Environment

```
def round(self, configuration, prices, alpha_ratios, item_sold_mean):
    prices_configuration = np.zeros(self.n_prod)
    for product in range(self.n_prod):
        prices_configuration[product] = prices[product][configuration[product]]
    simulator = SocialInfluence(self.lambda_coeff, alpha_ratios, item_sold_mean,
    ↪  self.customer_class, prices_configuration, self.n_prod, configuration)
    simulator.simulation()
    return simulator.reward, simulator.bought, simulator.actual_users
```

NonStationaryEnvironment

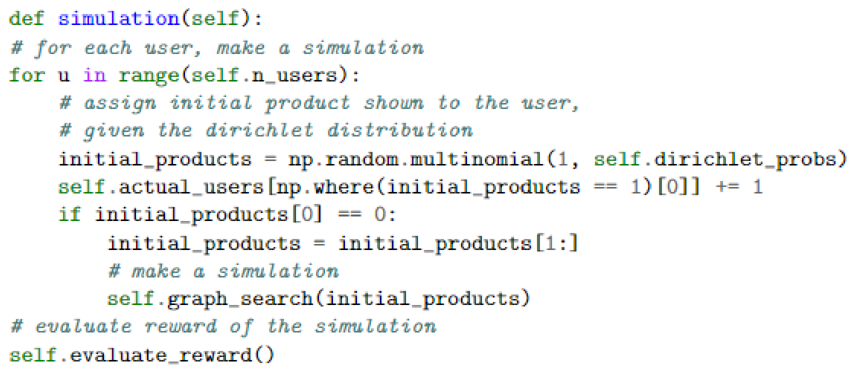
```
def round(self, configuration, prices, alpha_ratios, item_sold_mean, phase):
    prices_configuration = np.zeros(self.n_prod)
    for product in range(self.n_prod):
        prices_configuration[product] = prices[product][configuration[product]]
    simulator = SocialInfluence(self.lambda_coeff, alpha_ratios, item_sold_mean,
    ↪  self.customer_class, prices_configuration, self.n_prod, configuration)
    simulator.abrupt_simulation(phase)
    return simulator.reward, simulator.bought, simulator.actual_users
```

STEP 1 - Customer Class

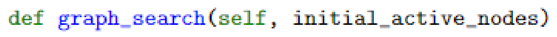


```
{
    "n_users": 100,
    "binary_features": [0, 0],
    "average_alphas": [0.02, 0.3, 0.01, 0.19, 0.3, 0.18],
    "graph_probabilities": [[0.0, 0.9, 0.01, 0.02, 0.07],
                             [0.25, 0.0, 0.25, 0.25, 0.25],
                             [0.5, 0.2, 0.0, 0.15, 0.15],
                             [0.8, 0.0, 0.1, 0.0, 0.1],
                             [0.1, 0.6, 0.1, 0.2, 0.0]],
    "conversion_rates": [[0.3, 0.4, 0.4, 0.3], [0.2, 0.3, 0.3, 0.3],
                         [0.4, 0.3, 0.2, 0.2], [0.2, 0.2, 0.5, 0.5],
                         [0.4, 0.2, 0.3, 0.4]],
    "reservation_prices": [[18, 15, 20, 30, 25]],
    "average_items_sold": [[3, 2, 5, 8], [2.1, 1.3, 0.6, 2.7],
                         [2, 3, 6, 8], [1, 7, 4, 4],
                         [0.7, 0.6, 1.4, 0.8]]
}
```

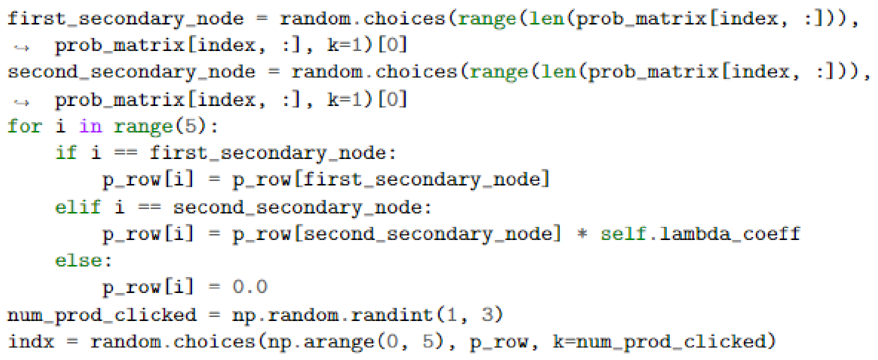
STEP 1 – Social Influence



```
def simulation(self):
# for each user, make a simulation
for u in range(self.n_users):
    # assign initial product shown to the user,
    # given the dirichlet distribution
    initial_products = np.random.multinomial(1, self.dirichlet_probs)
    self.actual_users[np.where(initial_products == 1)[0]] += 1
    if initial_products[0] == 0:
        initial_products = initial_products[1:]
        # make a simulation
        self.graph_search(initial_products)
# evaluate reward of the simulation
self.evaluate_reward()
```



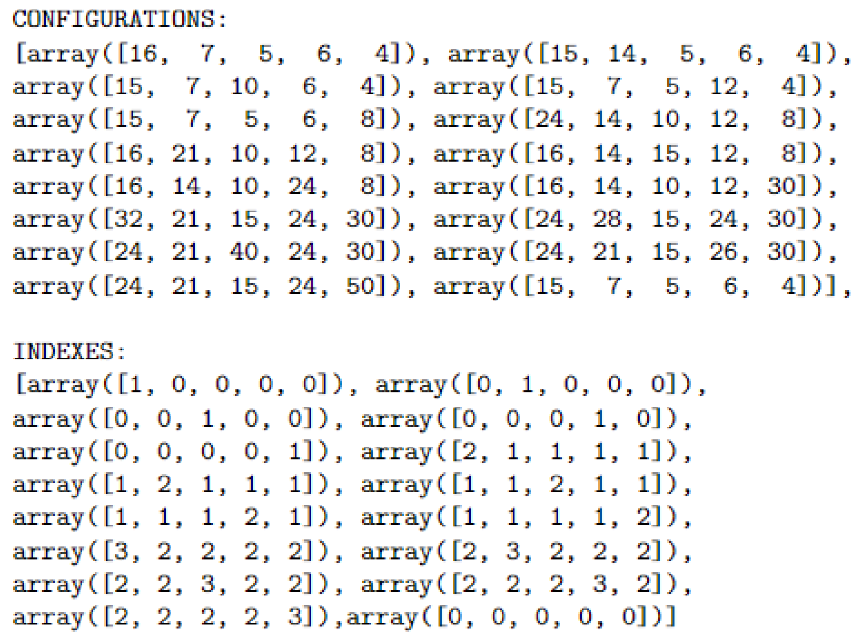
```
def graph_search(self, initial_active_nodes)
```



```
first_secondary_node = random.choices(range(len(prob_matrix[index, :])),
↪   prob_matrix[index, :], k=1)[0]
second_secondary_node = random.choices(range(len(prob_matrix[index, :])),
↪   prob_matrix[index, :], k=1)[0]
for i in range(5):
    if i == first_secondary_node:
        p_row[i] = p_row[first_secondary_node]
    elif i == second_secondary_node:
        p_row[i] = p_row[second_secondary_node] * self.lambda_coeff
    else:
        p_row[i] = 0.0
num_prod_clicked = np.random.randint(1, 3)
indx = random.choices(np.arange(0, 5), p_row, k=num_prod_clicked)
```

STEP 2 – Optimization Problem

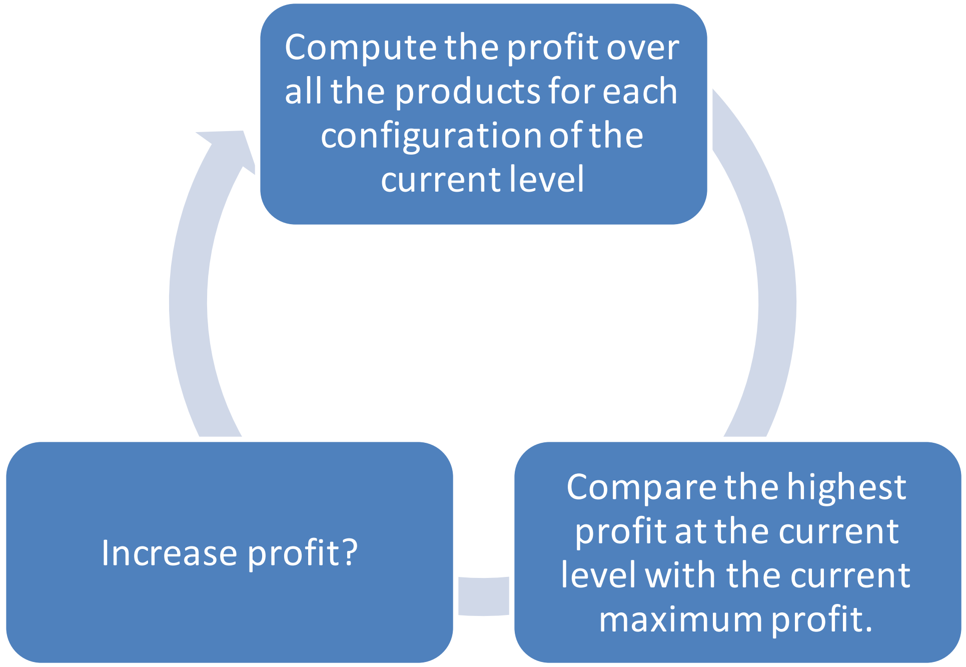
All the parameters are known and no online learner is required. The objective optimization problem is the maximization of the cumulative expected margin over all the products; in doing this, a greedy algorithm is used.



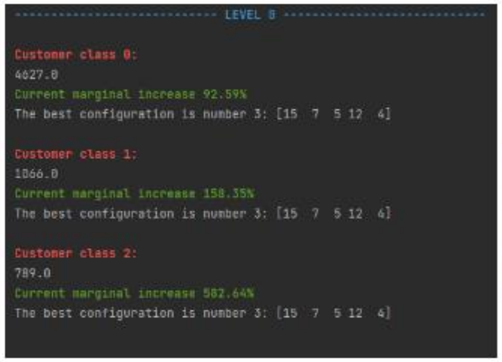
```
CONFIGURATIONS:
[array([16,  7,  5,  6,  4]), array([15, 14,  5,  6,  4]),
array([15,  7, 10,  6,  4]), array([15,  7,  5, 12,  4]),
array([15,  7,  5,  6,  8]), array([24, 14, 10, 12,  8]),
array([16, 21, 10, 12,  8]), array([16, 14, 15, 12,  8]),
array([16, 14, 10, 24,  8]), array([16, 14, 10, 12, 30]),
array([32, 21, 15, 24, 30]), array([24, 28, 15, 24, 30]),
array([24, 21, 40, 24, 30]), array([24, 21, 15, 26, 30]),
array([24, 21, 15, 24, 50]), array([15,  7,  5,  6,  4])],

INDEXES:
[array([1, 0, 0, 0, 0]), array([0, 1, 0, 0, 0]),
array([0, 0, 1, 0, 0]), array([0, 0, 0, 1, 0]),
array([0, 0, 0, 0, 1]), array([2, 1, 1, 1, 1]),
array([1, 2, 1, 1, 1]), array([1, 1, 2, 1, 1]),
array([1, 1, 1, 2, 1]), array([1, 1, 1, 1, 2]),
array([3, 2, 2, 2, 2]), array([2, 3, 2, 2, 2]),
array([2, 2, 3, 2, 2]), array([2, 2, 2, 3, 2]),
array([2, 2, 2, 2, 3]),array([0, 0, 0, 0, 0])]
```

STEP 2 – Optimization Algorithm

For every configuration level and for every user class:

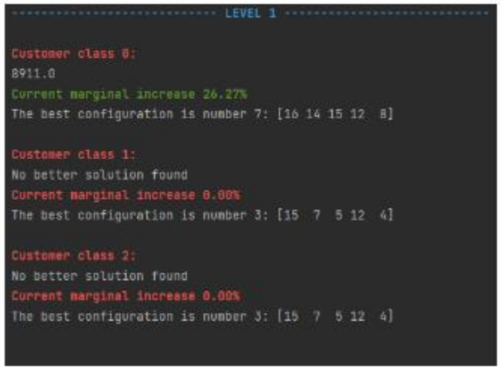

Compute the profit over all the products for each configuration of the current level
Compare the highest profit at the current level with the current maximum profit.
Increase profit?

STEP 2 – Results



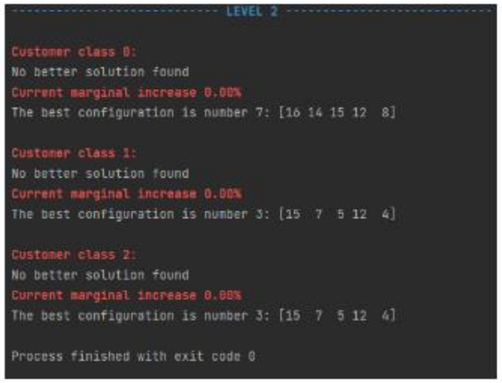
```
---------------------------- LEVEL 0 ----------------------------

Customer class 0:
4627.0
Current marginal increase 92.59%
The best configuration is number 3: [15  7  5 12  4]

Customer class 1:
1066.0
Current marginal increase 158.35%
The best configuration is number 3: [15  7  5 12  4]

Customer class 2:
789.0
Current marginal increase 582.64%
The best configuration is number 3: [15  7  5 12  4]
```

Figure 1: Algorithm Result (1)



```
---------------------------- LEVEL 1 ----------------------------

Customer class 0:
8911.0
Current marginal increase 26.27%
The best configuration is number 7: [16 14 15 12  8]

Customer class 1:
No better solution found
Current marginal increase 0.00%
The best configuration is number 3: [15  7  5 12  4]

Customer class 2:
No better solution found
Current marginal increase 0.00%
The best configuration is number 3: [15  7  5 12  4]
```

Figure 2: Algorithm Result (2)

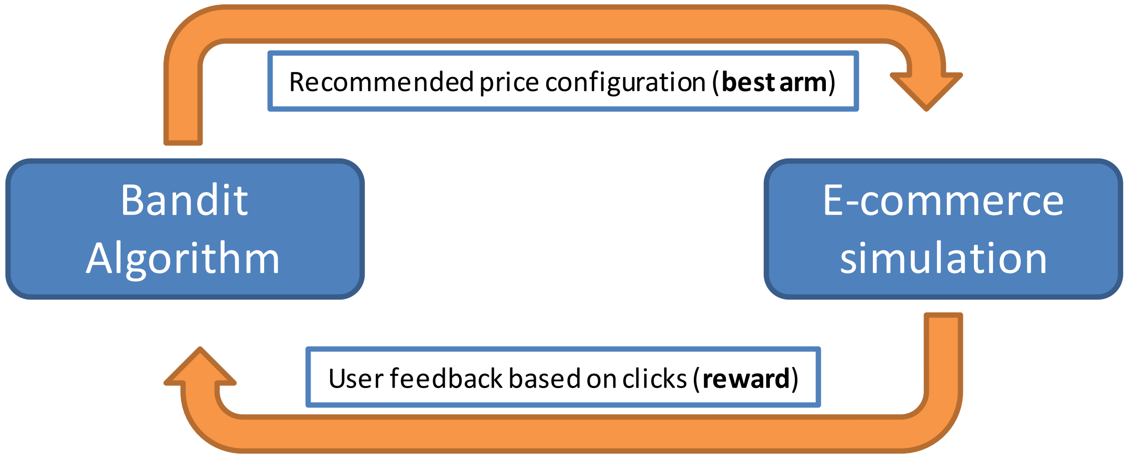


```
---------------------------- LEVEL 2 ----------------------------

Customer class 0:
No better solution found
Current marginal increase 0.00%
The best configuration is number 7: [16 14 15 12  8]

Customer class 1:
No better solution found
Current marginal increase 0.00%
The best configuration is number 3: [15  7  5 12  4]

Customer class 2:
No better solution found
Current marginal increase 0.00%
The best configuration is number 3: [15  7  5 12  4]

Process finished with exit code 0
```

Figure 3: Algorithm Result (3)

STEP 3 - Optimzation with uncertain conversion rates

Step 3 provides the implementation of *TS* and *UCB* bandit algorithms in the scenario where *binary features are unknown*, and thus classes are to be considered aggregate, and the algorithms have to estimate the *conversion rates*.


Recommended price configuration (**best arm**)
Bandit Algorithm
E-commerce simulation
User feedback based on clicks (**reward**)

The simulation of the e-commerce website has a duration of 200 days, and the number of experiments is 5.

In this step we also developed the clairvoyant estimator, which represents the best obtainable reward, and it is used to evaluate the regret.

STEP 3 - Thompson Sampling and UCB

Thompson Sampling: it exploits prior knowledge to choose the best arm which maximizes the expected reward based on a randomly drawn belief.

Beta distribution is initialized uniformly to a matrix of ones. From this distribution, the converison rates are evaluated for each configuration, and the one with the highest value is pulled. Then, simulation is run and the reward is collected. The update rule has been implemented as follows:

$$(\alpha_a, \beta_a) \leftarrow (\alpha_a, \beta_a) + (x_{a,t}, 1 - x_{a,t}) \xrightarrow{\text{Python}}$$

```
self.beta_parameters[product, pulled_config[product], 0] =
↪   self.beta_parameters[product, pulled_config[product], 0] +
↪   bought[product]
self.beta_parameters[product, pulled_config[product], 1] =
↪   self.beta_parameters[product, pulled_config[product], 1] + total -
↪   bought[product]
```

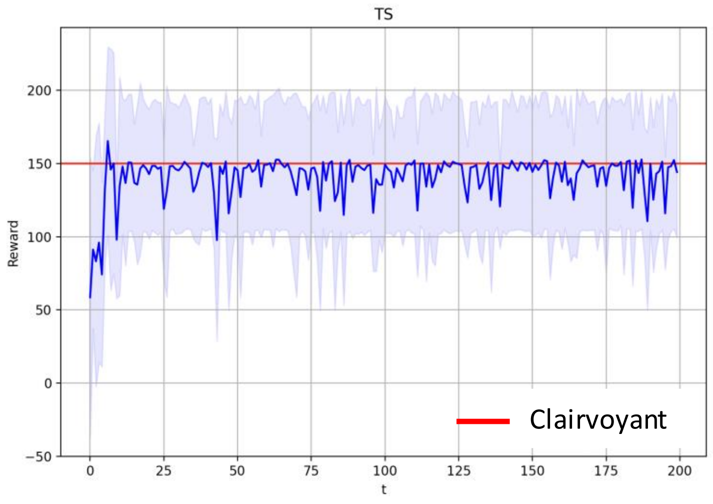
UCB-1: deterministic algorithm, where a confidence bound is associated to each arm. Confidence bound is updated according to the following formula:

$$a_t \leftarrow \arg\max_{a \in A} \left\{ \bar{x}_a + \sqrt{\frac{2\log(t)}{n_a(t-1)}} \right\} \xrightarrow{\text{Python}}$$

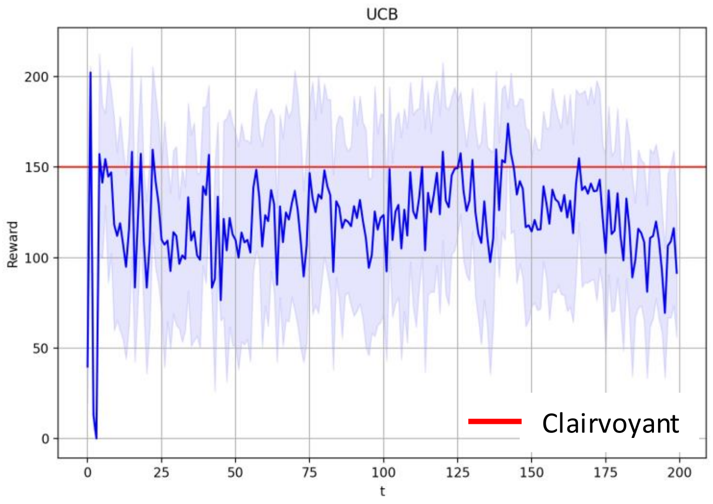
```
self.means[product, pulled_config[product]] = (self.means[product,
↪   pulled_config[product]] * seen[product, pulled_config[product]] +
↪   bought[product]) / tot_seen[product, pulled_config[product]]
self.upper_bound[product, pulled_config[product]] = m.sqrt((2 *
↪   m.log10(tot_samples[product])) / tot_seen[product,
↪   pulled_config[product]])
def pull_arm(self):
  pulled_config_indexes = np.argmax(self.means + self.upper_bound, axis=1)
  return pulled_config_indexes
```

Where the means are initialized to a matrix with all values equal to zero and the upper bound matrix to infinity.

STEP 3 - Results: reward

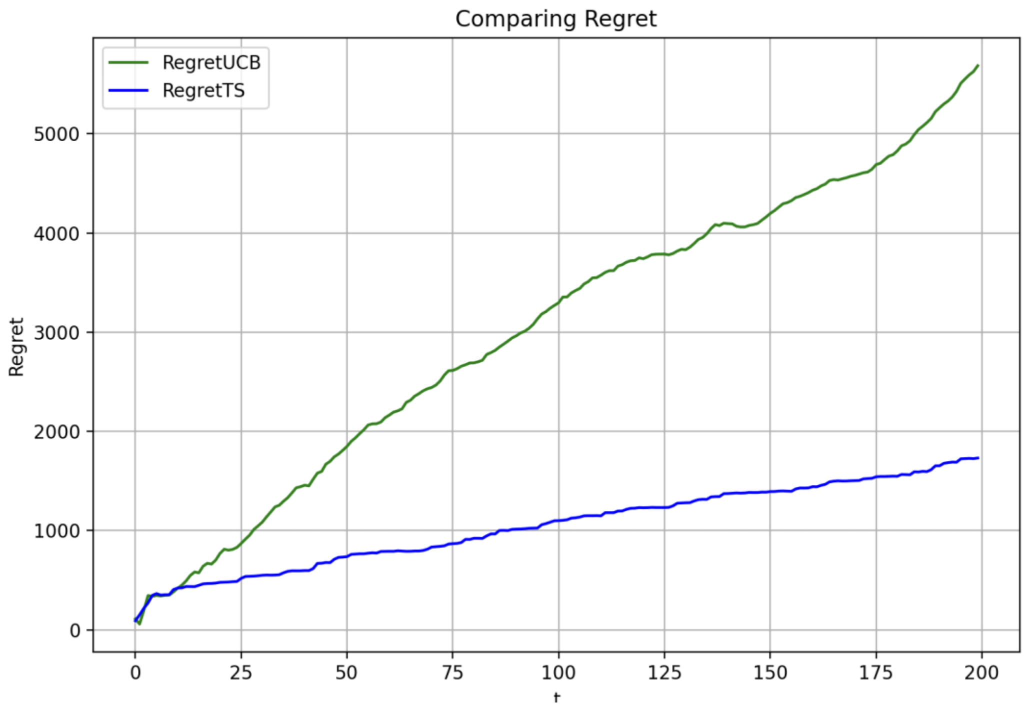


TS reward



UCB reward

STEP 3 - Results: regret



Regret comparison between UCB and TS algorithms

STEP 4 - *Optimization with uncertain conversion rates, α ratios, and number of items sold per product*

This step reproduces the same conditions as the previous one. The only difference is that now also the alpha ratios and the number of products sold are unknown:

α ratios:

```
def estimate_alpha_ratios(old_starts, starts):
    new_starts = old_starts + starts
    return new_starts / np.sum(new_starts)
```

Number (mean) of items sold per product:

```
def estimate_items_for_each_product(mean, seen_since_day_before, unit_sold,
↪   total_sold):
    if mean * seen_since_day_before + unit_sold == 0:
        return 0.
    return (mean * seen_since_day_before + unit_sold) / total_sold
```

STEP 4 - *Reward*

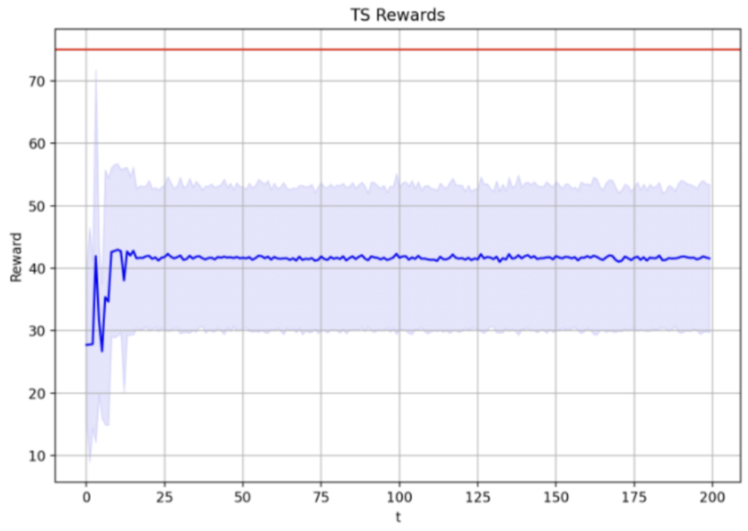


Figure 7: TS reward

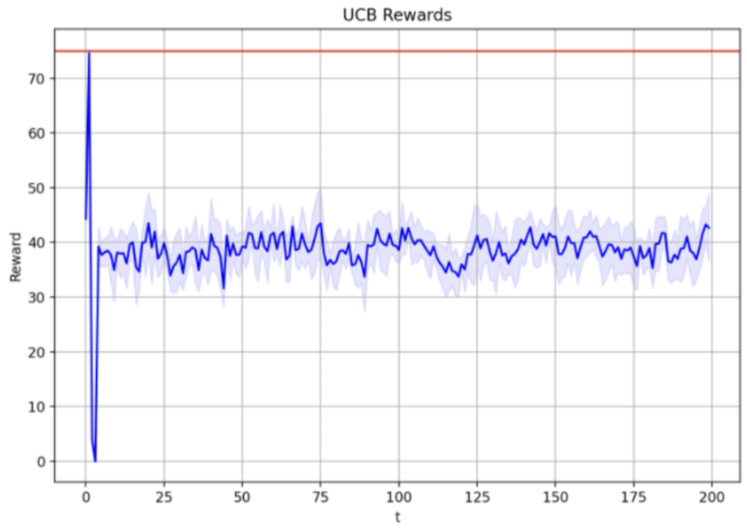


Figure 8: UCB reward

STEP 4 - *Cumulative regret*

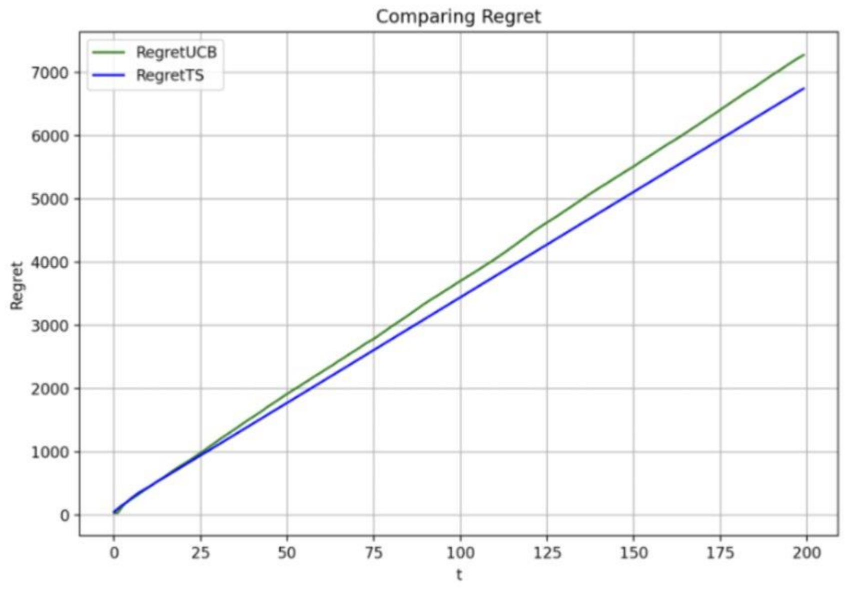


Figure 9: Cumulative regret

STEP 5 – Graph Search

The function **Graph Search** is useful to simulate:

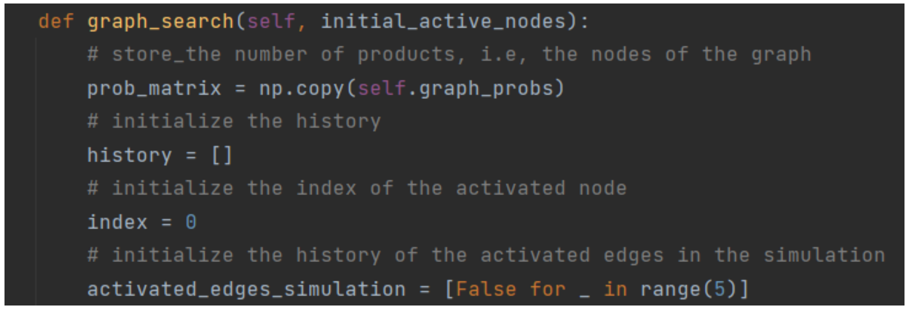
- How the products appear in the user page (e.g primary and secondary product shown).
- How the user interacts with the products.

The function is also relevant to collect data about:

- The number of clicks on the products.
- The number of units sold per product.
- The number of seen items per product.

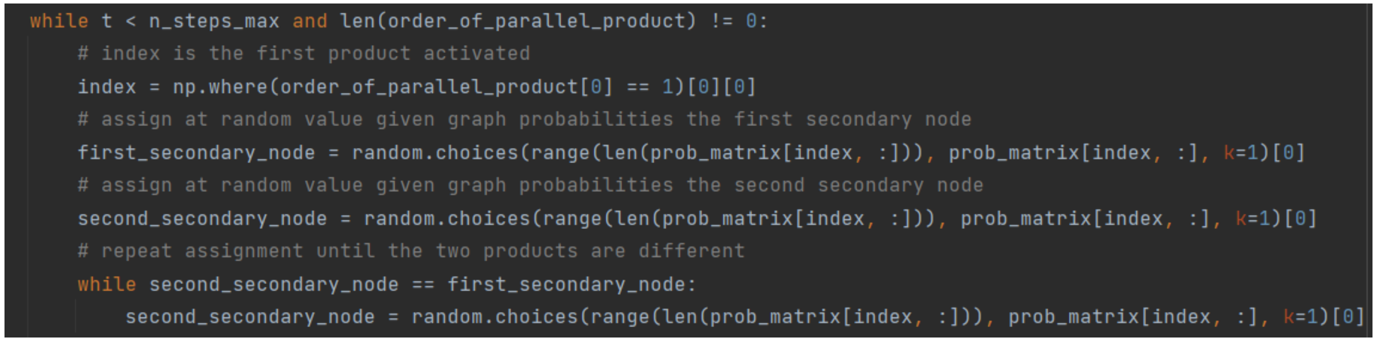
STEP 5 – Graph Search

At the beginning of the function we initialize the **Graph Probability Matrix**



```python
def graph_search(self, initial_active_nodes):
    # store_the number of products, i.e, the nodes of the graph
    prob_matrix = np.copy(self.graph_probs)
    # initialize the history
    history = []
    # initialize the index of the activated node
    index = 0
    # initialize the history of the activated edges in the simulation
    activated_edges_simulation = [False for _ in range(5)]
```

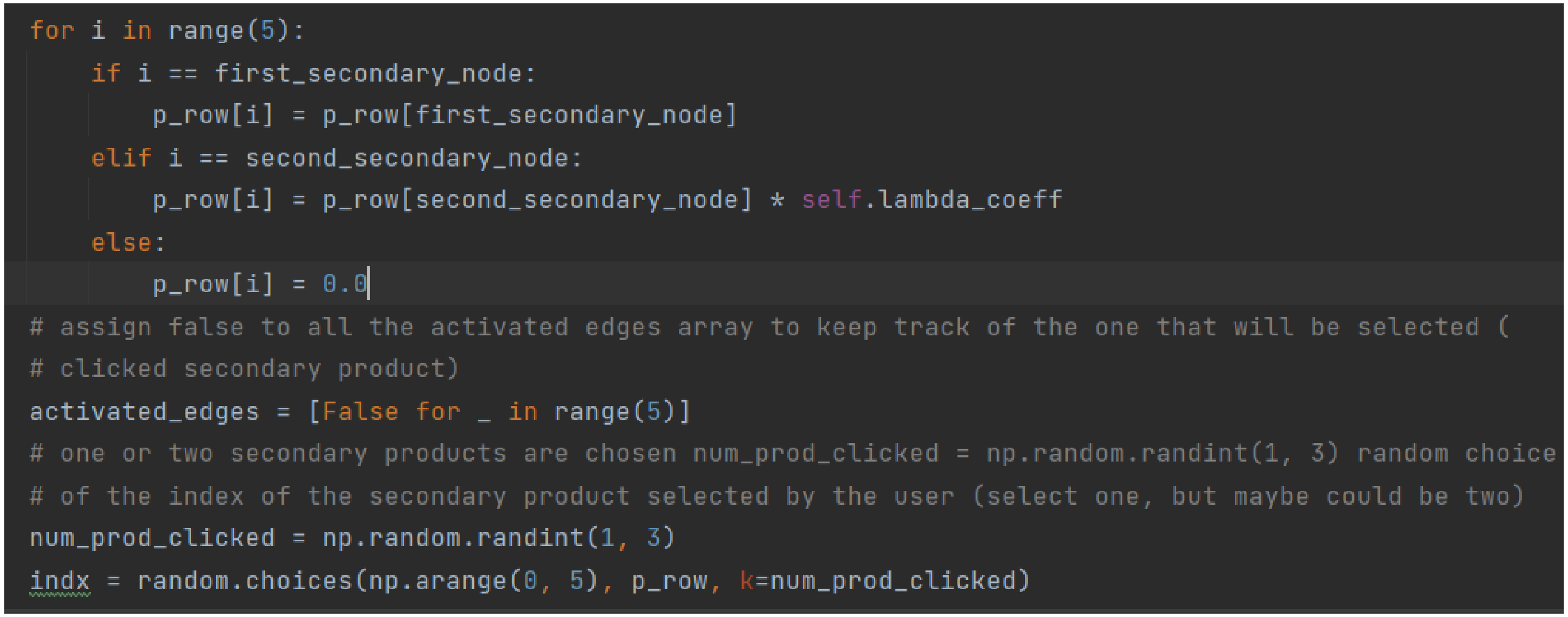
Then, we select the first and secondary product to be shown according to a **Probability Graph distribution** starting from the product selected by the user



```python
while t < n_steps_max and len(order_of_parallel_product) != 0:
    # index is the first product activated
    index = np.where(order_of_parallel_product[0] == 1)[0][0]
    # assign at random value given graph probabilities the first secondary node
    first_secondary_node = random.choices(range(len(prob_matrix[index, :])), prob_matrix[index, :], k=1)[0]
    # assign at random value given graph probabilities the second secondary node
    second_secondary_node = random.choices(range(len(prob_matrix[index, :])), prob_matrix[index, :], k=1)[0]
    # repeat assignment until the two products are different
    while second_secondary_node == first_secondary_node:
        second_secondary_node = random.choices(range(len(prob_matrix[index, :])), prob_matrix[index, :], k=1)[0]
```

STEP 5 – Graph Search

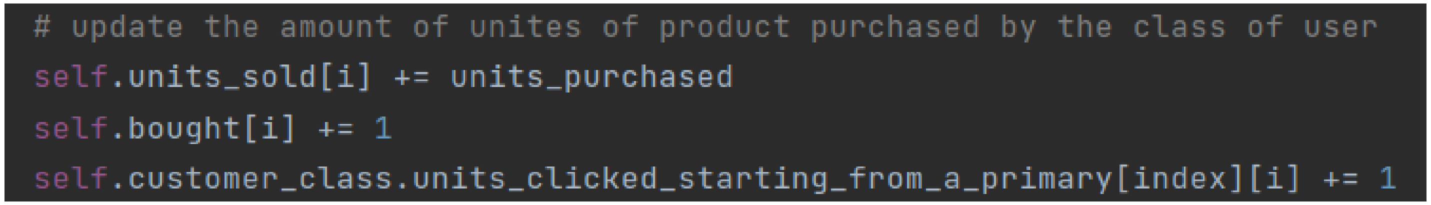
Other essential step to decide the user choice on the next product clicked



```python
for i in range(5):
    if i == first_secondary_node:
        p_row[i] = p_row[first_secondary_node]
    elif i == second_secondary_node:
        p_row[i] = p_row[second_secondary_node] * self.lambda_coeff
    else:
        p_row[i] = 0.0
# assign false to all the activated edges array to keep track of the one that will be selected (
# clicked secondary product)
activated_edges = [False for _ in range(5)]
# one or two secondary products are chosen num_prod_clicked = np.random.randint(1, 3) random choice
# of the index of the secondary product selected by the user (select one, but maybe could be two)
num_prod_clicked = np.random.randint(1, 3)
indx = random.choices(np.arange(0, 5), p_row, k=num_prod_clicked)
```

STEP 5 – Graph Search

Data Collection

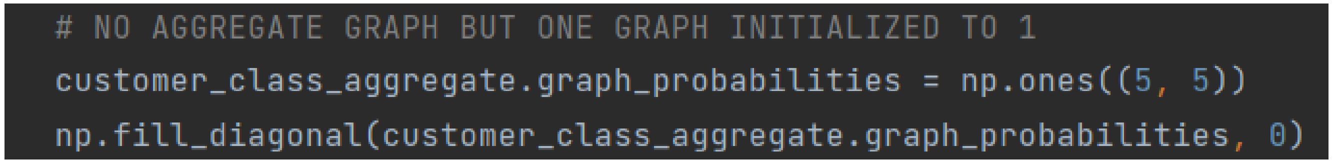


```
# update the amount of unites of product purchased by the class of user
self.units_sold[i] += units_purchased
self.bought[i] += 1
self.customer_class.units_clicked_starting_from_a_primary[index][i] += 1
```

Units_clicked_starting_from_a_primary is a matrix useful to keep track of the number of times a product is clicked starting from a given product (i.e defined by index).

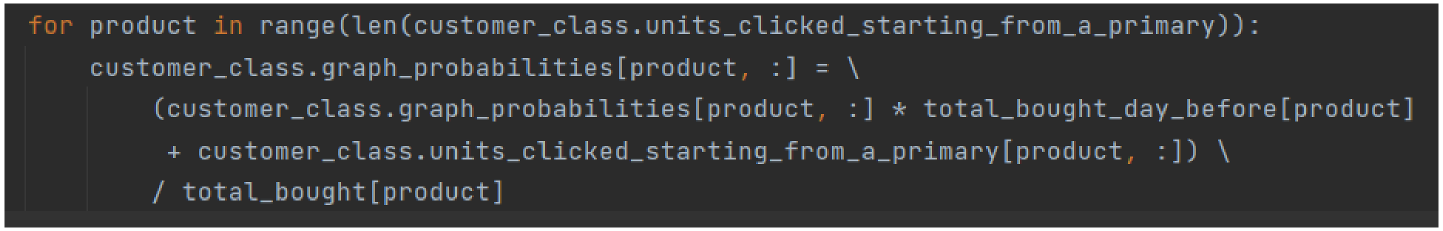
STEP 5 – Update Graph Probability

We keep the assumption of the previous steps, then we set the **Aggregate Graph Probability** to 1 for each cell of the matrix, except for the diagonal.



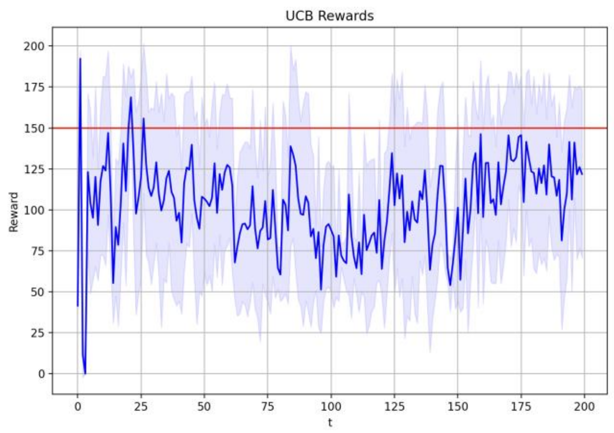
```
# NO AGGREGATE GRAPH BUT ONE GRAPH INITIALIZED TO 1
customer_class_aggregate.graph_probabilities = np.ones((5, 5))
np.fill_diagonal(customer_class_aggregate.graph_probabilities, 0)
```

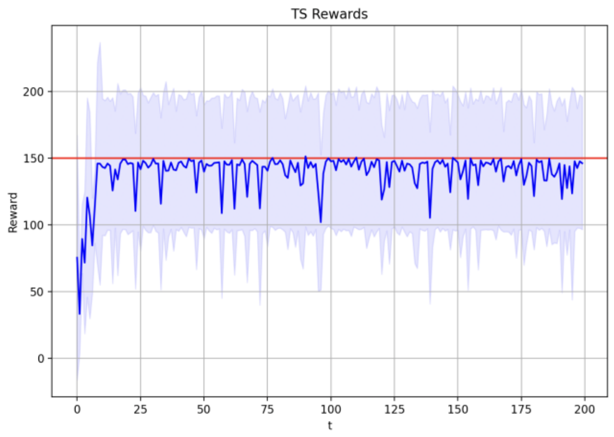
After each simulation we update the values of the **Aggregate Graph Probability** for each product

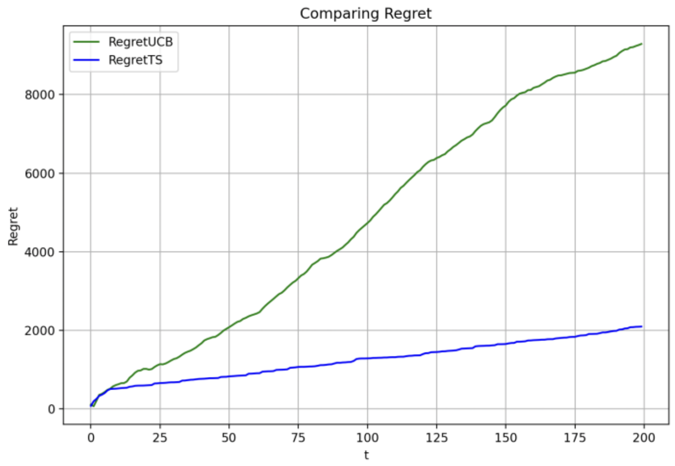


```
for product in range(len(customer_class.units_clicked_starting_from_a_primary)):
    customer_class.graph_probabilities[product, :] = \
        (customer_class.graph_probabilities[product, :] * total_bought_day_before[product]
         + customer_class.units_clicked_starting_from_a_primary[product, :]) \
        / total_bought[product]
```

STEP 5 – Results





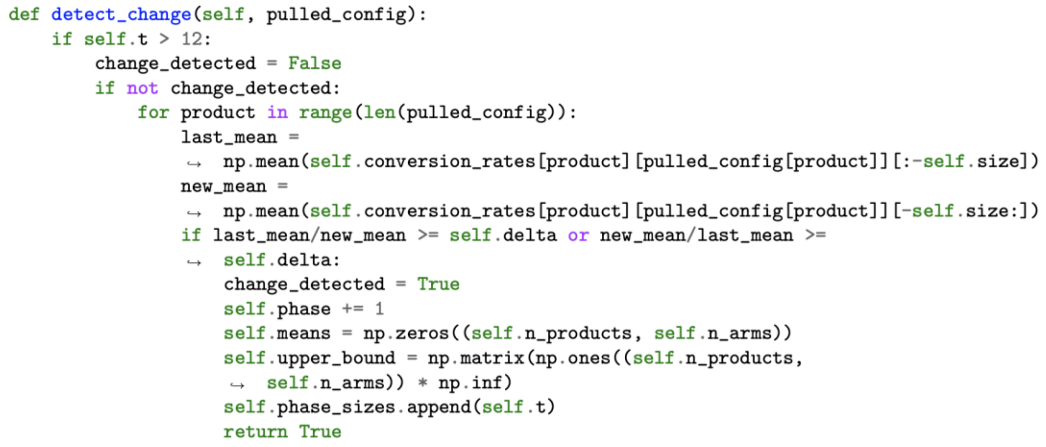


Step 6 - Non-stationary demand curve

Step 6 provides the implementation of two different *UCB-like algorithms*, which are able to tackle the same problem of step 3, but also with abrupt changes in demand curve.

In order to properly implement the *Change Detection algorithm* and the *Sliding Window* one, we developed a *NonStationaryEnviroment* module and the e-commerce simulation was modified to deal with changes. Also, the *clairvoyant was modified* such that it is able to evaluated the best reward for each phase.

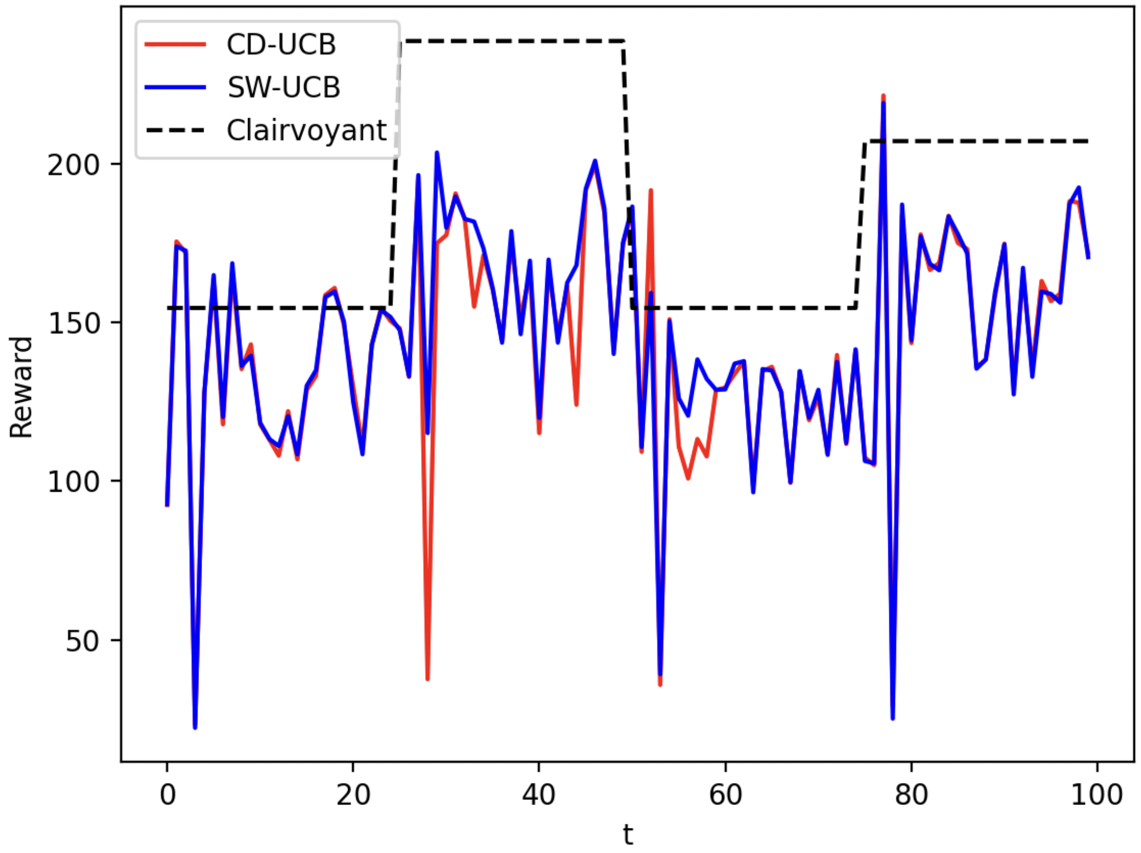
The difference between the Change Detection algorithm and the Sliding Window one resides in the fact that the former detects the variation in the demand curve, while latter one has a fixed window size which flows along the timestamps of the experiment.



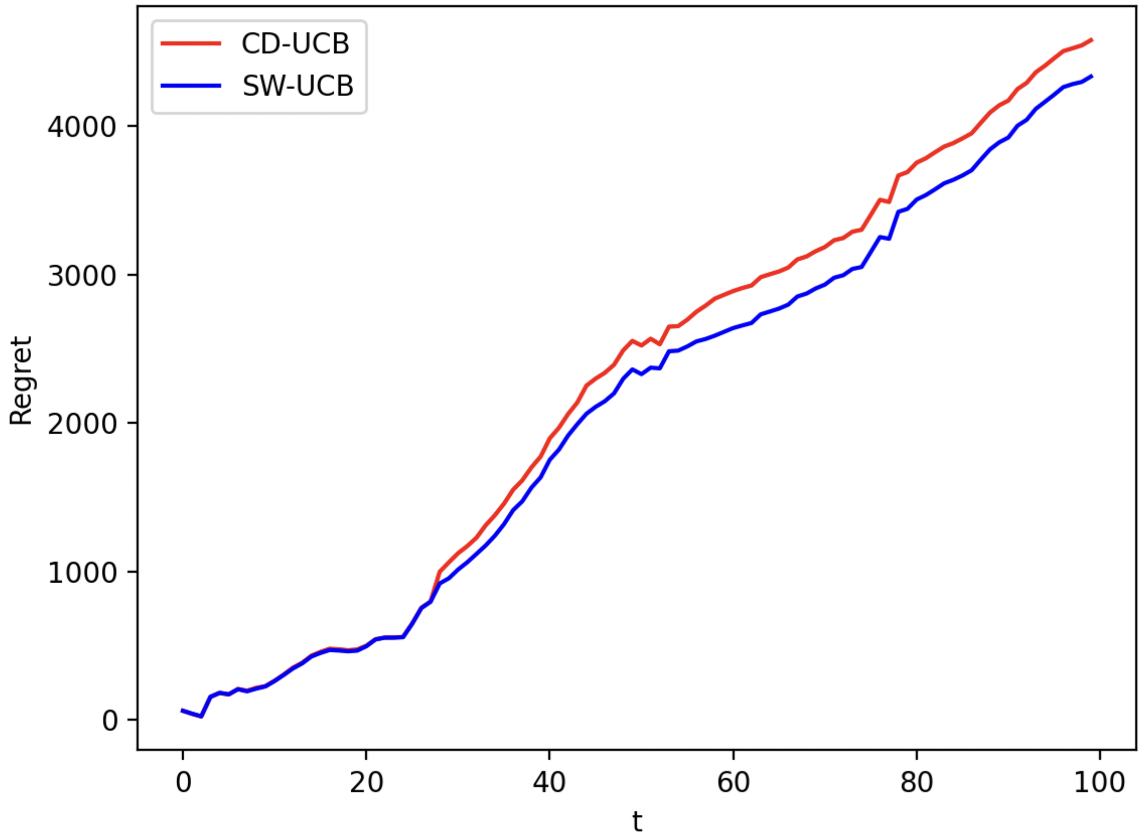
```
def detect_change(self, pulled_config):
    if self.t > 12:
        change_detected = False
        if not change_detected:
            for product in range(len(pulled_config)):
                last_mean =
                ↪   np.mean(self.conversion_rates[product][pulled_config[product]][:-self.size])
                new_mean =
                ↪   np.mean(self.conversion_rates[product][pulled_config[product]][-self.size:])
                if last_mean/new_mean >= self.delta or new_mean/last_mean >=
                ↪   self.delta:
                    change_detected = True
                    self.phase += 1
                    self.means = np.zeros((self.n_products, self.n_arms))
                    self.upper_bound = np.matrix(np.ones((self.n_products,
                    ↪   self.n_arms)) * np.inf)
                    self.phase_sizes.append(self.t)
                    return True
```

Implementation of the change detection function

Step 6 - Results: rewards

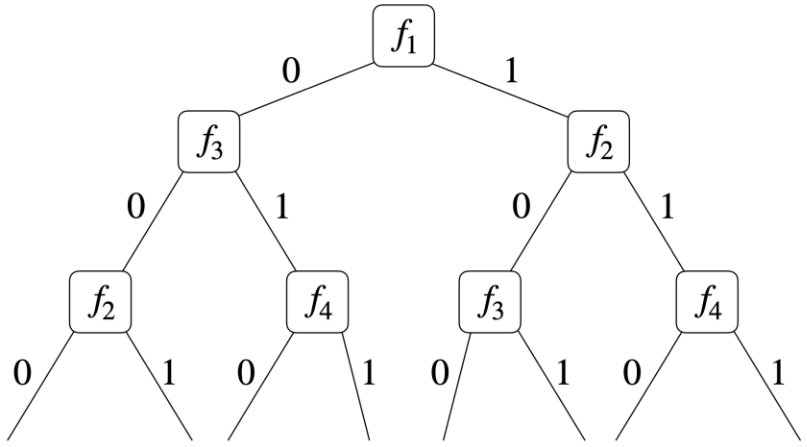


Step 6 - Results: regret



Step 7 - Context generation

The starting point of the following chapter is step number 4: conversion rates, alpha ratios and the number of items sold per product are unknown but this time the features can be observed by the e-commerce website. Every 14 days a Context Generation algorithm is executed to build a new features tree: we need to evaluate every possible partition of the space of the features, and for every one of these, we need to evaluate whether dis-aggregating such a subset is better than abstaining to do that. Practically, the features tree defines which context to use.



Step 7 - Context generation

The algorithm is initialized with "no feature" classes which aggregate all the binary features. These classes - as the algorithm proceeds - are disaggregated and, after the two learners are trained on the initial partitions of data, a condition is evaluated and, if it is satisfied, the node is also disaggregated and the algorithm proceeds recursively. The splitting condition is as follows:

$$p_l\mu_l + p_r\mu_r \geq \mu \quad \xrightarrow{\text{Python}}$$

```python
def evaluate_split_condition(self, rewards0, rewards1, time):
    check = False
    l_lowerbound = self.lower_bound(rewards0, 5, 14)
    r_lowerbound = self.lower_bound(rewards1, 5, 14)
    p = self.assign_prob_context_occur(time)
    if p * (l_lowerbound + r_lowerbound) >= self.father_lower_bound:
        check = True
    return check, l_lowerbound, r_lowerbound
```

Where μ, μl and μr are the expected rewards, pl and pr are the probability weights and the lower bound μ is:

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

```python
def lower_bound(self, rewards, confidence, cardinality):
    return np.mean(rewards) - np.sqrt(-np.log10(confidence)/(cardinality * 2))
```

Where x is the expected reward computed by simulating the learner and |Z| is the size of the data.

Step 7 - *Reward*

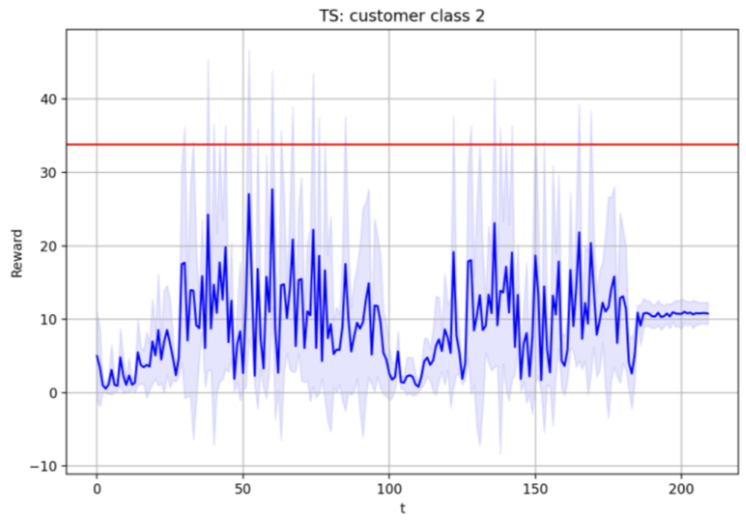


Figure 21: TS reward

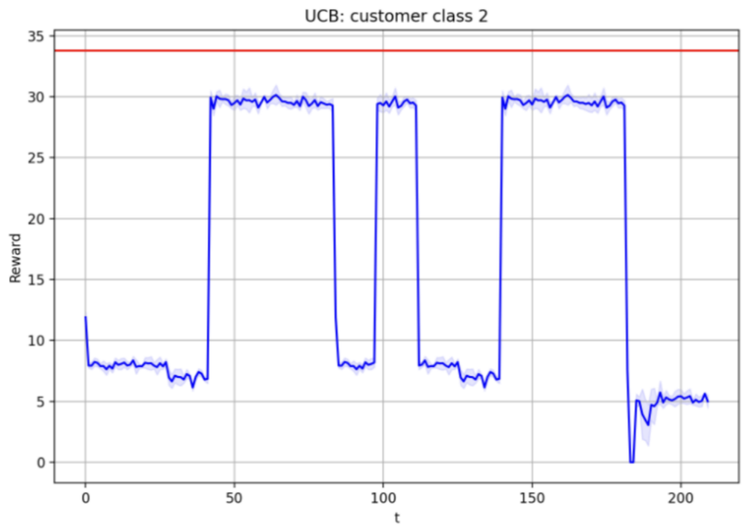


Figure 22: UCB reward

Step 7 – *Cumulative Regret*

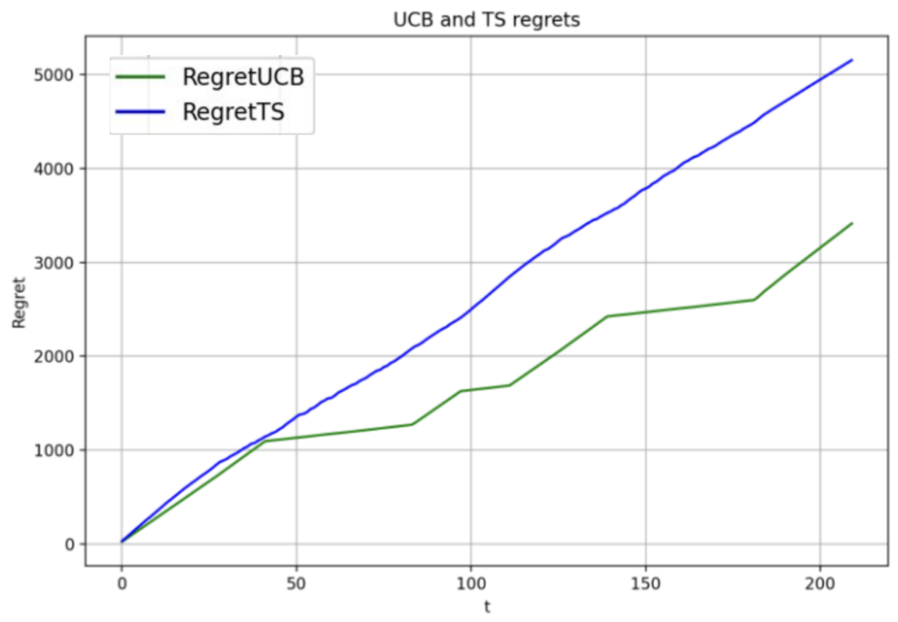


Figure 23: Cumulative regret

POLITECNICO
MILANO 1863