Online Learning Applications
Academic Year 2021 - 2022

Pricing

Marco Tonnarelli Alice Brugnoli Andrea Isgrò
Mattia Sabella Giovanni Alessandro Clini

Professor

Nicola Gatti

Contents

Introduction

This paper deals with e-commerce trying to establish the optimal price for 5 different products without any storage cost. The aim of e-commerce is to maximize its revenue; in doing this, we need the demand curve of given products which is a graph depicting the relationship between the price of a certain commodity (the y-axis) and the quantity of that commodity that is demanded at that price (the x-axis). In real-world applications demand curves are not available a-priopri therefore they need to be estimated via online learning techniques such as *Thompson Sampling, Upper Confidence Bound (UCB) Bandit Algorithm etc.*

The website is characterized as it follows: once the user enters our website all the products are displayed and then, once he clicks on one of the products, a web page is created in which the clicked product, called primary, is shown with its price. If this product is added to the cart by the user, then two other products, whose prices are hidden, are recommended by the system in different slots ordered by relevance. If the user clicks one of them, called secondary, then a new page is loaded in which the clicked one is displayed together with its own price. The probability to look at the second slot is lower with respect to the first one which is 1 considering the corresponding primary product added to the cart. The number of visited pages is finite since the system will not show products which have already been seen. In the end, the customer buys what is in the cart. Furthermore, each customer is characterized by a personal reservation price which prevents him from buying a single unit of a product if the price is higher than that.

We imagine that every day a random number of potential customers access one of the e-commerce pages structured as above with a certain probability.

1 Step 1: Environment

Our environment is made up of a set of classes in which we define the main entities which contribute to the life cycle of our e-commerce and their interactions. The main task of the environment is to advance the learning algorithms in the time domain through the function *round* which simulates the social context and returns, in all cases, the rewards, the units sold, and the units seen by the customers.
The actual environment is composed of the following classes:

- **Environment:** it is the main component; It handles the stationary case.
- **NonStationaryEnvironment:** it is a specific realization of the environment, related to the non stationary case, which is caused by demand curves abrupt changes, in the sixth step.
- **ContextualEnvironment:** it aims to manage the non-aggregated user classes in the seventh step.

Then there are also other two classes which are used in the scope of the environment:

- **CustomerClass:** in here we define the attributes of a single customer with respect to the belonging class.
- **ContextClass:** it aims to manage scenarios in the seventh step and it also implements the *Context Generation* algorithm.

1.1 Classes

In the base environment class, we define the initialization and the *round* method which simulates a step in the algorithm and represents the experience of the users in a day.

The parameters we define are as follows:

- **n_arms:** it is the cardinality of the set of possible prices.
- **customer_class:** it is an object representing a particular customer class.
- **lambda_coeff:** it is a probability's decay coefficient of seeing the second slot given the primary product. It is known a priori.
- **n_prod:** it is the cardinality of the set of available products.

The method *round* is defined as follows:

```
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
```

In the *NonStationaryEnvironment* class, we define the same parameters and method. In addition, we add a parameter t which represents the time, and a parameter *phase* which defines the different phases in the *round* method. This method is reported in the following:

```
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
```

The *ContextualEnvironment* slightly differs from the other environment: it is initialized with no CustomerClass and it has a method called *select_customer_class*, the setter for the attributes *CustomerClass*.

In the end, in the **CustomerClass** we define the initialization of the *CustomerClass* object. The parameters we initialize are the following:

- **conversion_rates:** it is a 5x4 matrix - initialized at $\overline{0}$ - containing the probability of a user buying a product at a certain price. It can be known or unknown.
- **number_of_customers:** it represents the number of customers of the particular class.
- **alpha_probabilities:** it is a vector of six elements - initialized at $\overline{0}$ - which represents the probabilities for the user to enter the e-commerce for the first time with the corresponding product as primary or to end up in a different website.
- **item_sold_mean:** it is a matrix 5x4 containing the average number of sold products with a certain price. It is used for generating the actual number of items each user buys during the simulation.
- **reservation_prices:** it is a 5 elements vector containing, for a particular customer class, the maximum amount of money the user is willing to spend for a particular product.
- **units_clicked_starting_from_a_primary:** it is a 5x5 matrix whose aim is to save the number of clicks on a certain product when starting from a specific primary node.
- **graph_probabilities:** it is a 5x5 matrix containing the probabilities for a user to click on a product *p2* while displayed in the first slot when *p1* is displayed as primary.

Most of these parameters are first initialized as empty, then, when needed, *DataManager* is called to fill them in order to make them ready to be used. The *DataManager* file contains different methods, the ones used to get the *CustomerClass* take as input some JSON files. These files contain information about a single customer class. For example, the JSON file for the first user class is reported below:

```
    {
  "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]]
}
```

In these files, features of a particular user class are reported. In particular, we define the **binary_features** that distinguish each class of users: we can suppose that the first feature represents male/female users while the second one young/old users. Similarly, we retrieved the product prices from a *json* file:

```
   {"prices": [
  [7, 10, 24, 32],
  [5, 14, 14, 28],
  [5, 10, 15, 20],
  [15, 12, 24, 29],
  [4, 8, 37, 50]]
}
```

1.2 Environment Applications

The *Enviroment* object is created in steps number 3, 4 and 5 in order to obtain all the results needed after the simulation of a single day through the method *round*. It retrieves specific Social Influence methods: *Simulation*, *Evaluate_reward* and *Graph_search* which will be discussed in depth later. The same happens for *NonStationaryEnvironment* and *ContextualEnvironment* respectively in step 6 and step 7.

1.3 Social Influence

The life-cycle of e-commerce and its interactions with the users is modelled by the *SocialInfluence* class. The *SocialInfluence* object is created in the same way in all the environment types as *Simulator*.

```
simulator = SocialInfluence(self.lambda_coeff, alpha_ratios, item_sold_mean,
↪  self.customer_class, prices_configuration, self.n_prod, configuration)
```

Then the *simulation* method is called and it simulates the user experience in the e-commerce through a graph-search algorithm given the requirements.

```
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()
```

Its functioning consists in initializing the *initial_products* vector whose size is 6. This vector will have the *i-th* element equal to 1 to indicate whether the user has opened the e-commerce website with the *i-th* product displayed as primary or they ended up in another website. This process starts from the α *ratios* which were obtained through a *Dirichlet* distribution whose parameters are retrieved from the json files previously mentioned, and they are different with respect to the user class.

```
alpha_ratios = np.random.dirichlet(customer_class.alpha_probabilities)
```

The *Dirichlet* distribution is a multivariate continuous probability distribution often used to model the uncertainty about a vector of unknown probabilities. The probability density function p of a Dirichlet-distributed random vector X is proportional to:

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1}, \tag{1}$$

where x are vectors that fulfil the following conditions:

$$x_i > 0, \sum_{i=1}^{k} x_i = 1, \tag{2}$$

Indeed values in the alpha_probabilities vector respect these constraints. Then the i-th cell is set to 1 according to a realization of a multinomial distribution, whose values are generated according to an instance of the Dirichlet distribution which in turn is generated according to the mean values present in "average_alphas."

If the user did not enter another website, the first column of the *initial_products* vector - which indicates the exit of the user from the e-commerce - is deleted and then the remaining vector is passed to the *graph_search* method which simulates the experience of that user. In the end, we evaluate the reward of the entire simulation through the *evaluate_reward* method. From here on out, a graph is considered: it represents the different scenarios that can arise from simulating users going through the website; it is also retrieved from json files and it is different with respect to the

user class. The Nodes are the products whereas the weights on the edges of the graph represent the probabilities with which a product can be displayed as secondary with respect to the primary. This information is represented in the python code as a 5x5 matrix. The simulation of the customer experience is carried out by the *graph_search* method:

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

and it works as follows: Firstly, the algorithm checks whether the first product displayed has a price higher than the reservation price; if the condition is true, the algorithm stops. The displayed price is taken by the corresponding *prices_configuration* pulled by the *Learner* algorithm.

```
if self.customer_class.reservation_prices[0][int(np.where(active_nodes == 1)[0])]
↪  < self.configuration[int(np.where(active_nodes == 1)[0])]:
    return
```

Then the algorithm, given the graph probabilities, displays to the user two secondary products,

```
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)
```

and randomly it lets the user decide to visit one or both secondary ones, which product to buy between them and it updates relevant attributes.

```
for i in range(5):
    # if the chosen secondary product is found, let the costumer actually buy it,
    ↪  by updated the
    # values related to the units sold and the reven
    if activated_edges[i] and self.customer_class.reservation_prices[0][i] >=
    ↪  self.configuration[i]:
        # assign a random amount of units of product purchased by the user
        units_purchased = np.random.randint(1, max(2,
        ↪  (self.item_sold_mean[i][self.configuration_indexes[i]]*2)))
        # 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
        # assign 1 to the new active nodes
        newly_active_nodes[i] = 1
```

In the end, the algorithm updates the transition probability of the new active nodes to zero value so that it is not possible to reach again the same node.

In the *SocialInfluence* class two other methods are defined: they regulate the user experience in the e-commerce when the environment is not stationary.

```
def abrupt_simulation(self, phase):
```

```
def abrupt_changes_graph_search(self, initial_active_nodes, phase):
```

They only differ from the previous one for one parameter. The parameter **phase** tells the algorithm in which phase it is operating.

2 Step 2: Optimization algorithm

In this step, we focus on the case in which we know all the parameters 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.

At first, in the *optimization_problem* method we initialize the parameters needed:

```
price_configurations, price_configuration_indexes, customers =
↪  initialization_step2(prices, number_of_configurations, classes)
```

The *initialization_step2* method is in the *generateData.py* module which contains several methods to support and manage data. It creates all the prices configuration with their indexes and initializes the three customer classes. The configurations of prices are created - given the requirements specifications and the prices - as follows:

```
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])]
```

Note that the 15th element corresponds to the configuration in which all the products have the lower price and it will be used as a starting point for comparing the profits of the new configurations. This is used in the initialization of the algorithm for computing the first reference maximum profit for the current customer class:

```
if level == 0:
    social = SocialInfluence(0.5, customers[customer_class].alpha_probabilities,
    ↪  customers[customer_class].item_sold_mean, customers[customer_class],
    ↪  price_configurations[15], number_of_products,
    ↪  price_configuration_indexes[15])
    social.simulation()
    for prod in range(number_of_products):
        maximum_profit[customer_class] += (price_configurations[15][prod] -
        ↪  margin[prod]) * social.units_sold[prod]
```

Then, for every configuration level and for every user class

```
for level in range(0, 15, 5):
    for customer_class in range(number_of_customer_classes):
```

1. Compute the profit over all the products for each configuration of the current level obtained by increasing, each time, the price of just one product of the original super arm.

   ```
   for i in range(5):
       # actual index
       idx = level + i
       # assign actual profit value (average)
       social = SocialInfluence(0.5,
       ↪  customers[customer_class].alpha_probabilities,
       ↪  customers[customer_class].item_sold_mean, customers[customer_class],
       ↪  price_configurations[idx], number_of_products,
       ↪  price_configuration_indexes[idx])
       social.simulation()
       for prod in range(number_of_products):
           profit[i] += (price_configurations[idx][prod] - margin[prod]) *
           ↪  social.units_sold[prod]
   ```

2. Compare the highest profit at the current level with the current maximum profit.

   ```
    if profit[possible_optimal[0][0]] > maximum_profit[customer_class] and
    ↪  check[customer_class]:
   ```

3. At this point there can be two different cases:

   (a) there is an increase in the profit: the maximum profit is updated and the algorithm is repeated for the reference customer class.

   (b) there is no increase for the reference customer class: the algorithm stops, returns the best configurations for the current customer class and eventually continues for the other customer classes.

2.1 Results

In the following, it is reported the result of an attempt in running the algorithm described above:

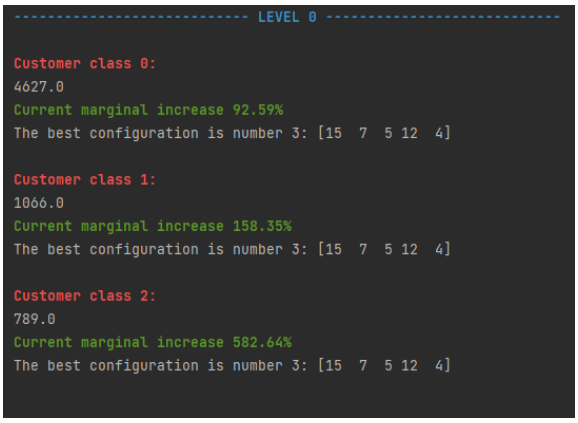

Figure 1: Algorithm Result (1)

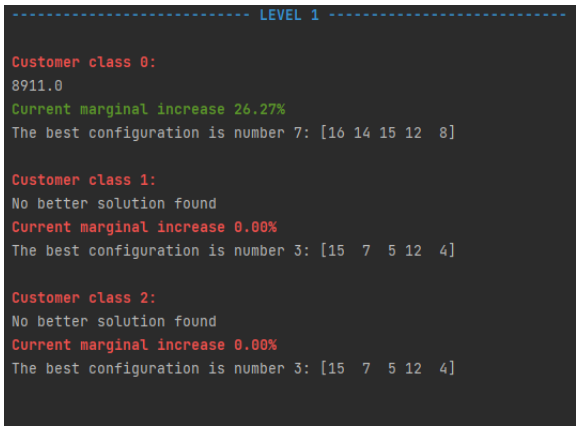

Figure 2: Algorithm Result (2)

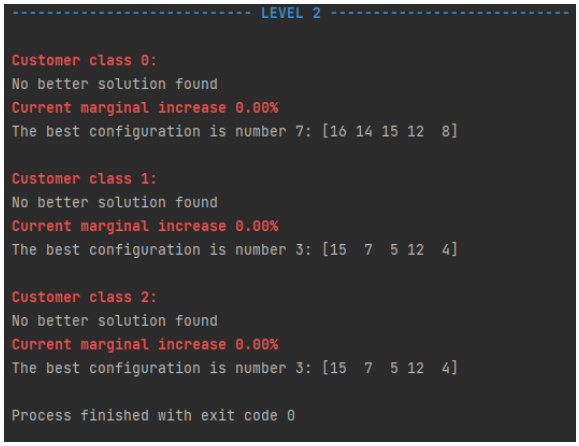

Figure 3: Algorithm Result (3)

As can be seen from the computation, the end result is suboptimal because the exploration of the various cost configurations, which are themselves of a limited number, ends at the point when a configuration of a later level is no better than that of the previous level. This is because the algorithm used is a greedy one, and the solution obtained will always be suboptimal because it does not explore all possible outcomes and the result is deterministic.

3 Step 3: Optimization with uncertain conversion rates

Now the case in which the binary features of the customer class are unknown and therefore the data is considered to be aggregated. Furthermore, the conversion rates of the customers are unknown. To overcome this issue, two different learners have been designed and implemented: *UCBLearner* and *TSLearner* respectively *Upper confidence bound learner* and *Thompson Sampling learner*. These two algorithms slightly differ from each other but both of them, for each day of the e-commerce activity, follow these steps:

- Select a super arm (configuration of prices)
- Run the *SocialInfluence* simulation to estimate relevant parameters
- Collect the rewards
- Update the algorithm parameters

Regarding the unknown binary features, the data of the three different user classes have been aggregated. This means that it has been created an aggregated user class whose attributes are formed by the weighted average of the attributes of the three different customer classes.
The number of days the e-commerce is analysed is set to 200. For each day of activity we set the number of experiments - the times the algorithms are run - to 5.

Moreover, it has been computed the Clairvoyant value, which represents the best obtainable reward, in order to make a comparison with the results obtained through the algorithms.

```
def evaluate_clairvoyant(configurations, max_units_sold, reservation_prices,
↪  n_users):
max_reward = 0.
for config in configurations:
    opt_reward = 0.
    # for each product
    for product in range(5):
        if reservation_prices[product] >= config[product]:
            # evaluate the reward for
            opt_reward += max_units_sold * n_users * config[product]
    # average reward over all the simulations
    opt_reward = opt_reward / n_users
    if opt_reward > max_reward:
        max_reward = opt_reward
return max_reward
```

3.1 UCB-1

The deterministic algorithm Upper Confidence Bound aims to associate an upper confidence bound to every arm and eventually to provide an optimistic estimation of the reward.

Through the UCB algorithm, the *Conversion Rates* have been estimated as the sum of the *mean* and the *upper confidence bound* of every single arm which are both initialized as follows:

```
self.means = np.zeros((n_products, n_prices))
self.upper_bound = np.matrix(np.ones((n_products, n_prices)) * np.inf)
```

Then, the best configuration is selected, pulled and used for interacting with the environment.

```
def pull_arm(self):
  pulled_config_indexes = np.argmax(self.means + self.upper_bound, axis=1)
  return pulled_config_indexes
```

The next step is to update the mean and the upper confidence parameters of the configuration that was pulled last time. It has been done in the following way:

```
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]])
```

where:

- **self.means[product, pulled_config[product]]:** it is the mean of the product with the price it has in the last pulled configuration.
- **seen[product, pulled_config[product]]:** it is the number of times the product with that price has been seen until the day before.
- **bought[product]:** is the number of times the product has been bought till now.
- **tot_seen[product, pulled_config[product]]:** it is *seen[product, pulled_config[product]]* plus the number of times it has been seen in the current day.
- **tot_samples[product]:** it is the number of times the product has been seen till now.

3.2 TS

The Thompson Sampling algorithm works as follows: for every arm, we have a prior on its expected value. In the case the arms' rewards are Bernoulli distribution, the priors are Beta distributions. (with the opportune parameters the Beta distribution is a uniform distribution). We draw a sample for every arm according to the corresponding Beta and take the one with the best sample. In the end, we update the Beta distribution of the chosen arm according to the observed realization and we do another iteration of the algorithm.
Given the following definitions:

$x_{a,t}$	Realization of random variable X_a at t
$\mathbb{P}(\mu_a = \theta_a)$	Prior on the expected value of X_a
θ_a	Variable of $\mathbb{P}(\mu_a = \theta_a)$
(α_a, β_a)	Parameters of Beta distribution $\mathbb{P}(\mu_a = \theta_a)$

We have that:
1) The Beta probability distribution is given by:

$$\mathbb{P}(\mu_a = \theta_a) = \frac{\Gamma(\alpha_a + \beta_a)}{\Gamma(\alpha_a)\Gamma(\beta_a)} (\theta_a)^{\alpha_a - 1} (1 - \theta_a)^{\beta_a - 1}$$

2) The Updating rule of the Beta distribution given a realization is:

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

The steps of the algorithm are reported above:

1. Set all the α and β parameters equal to one.

```
self.beta_parameters = np.ones((n_products, n_prices, 2))
```

2. Estimate the conversion rate by drawing a sample from a Beta distribution with α and β parameters and choose the super arm that returns the highest reward.

```
pulled_config_indexes = np.argmax(np.random.beta(self.beta_parameters[:, :,
↪  0], self.beta_parameters[:, :, 1]), axis=1)
```

3. Collect rewards through e-commerce simulation for all the users.

```
reward_ts, units_sold_ts, total_seen_ts =
↪  env.round(pulled_config_indexes_ts, prices, alpha_ratios,
↪  item_sold_mean)
```

4. Update the α and β parameters according to the observations of the environment.

```
for product in range(len(pulled_config)):
    self.update_observations(pulled_config[product], bought[product],
    ↪  product, reward)
    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]
```

5. Iterate the algorithm from step 2 until the last day.

3.3 Results

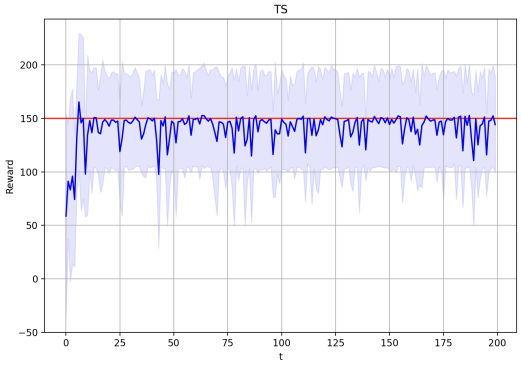

Figure 4: TS reward

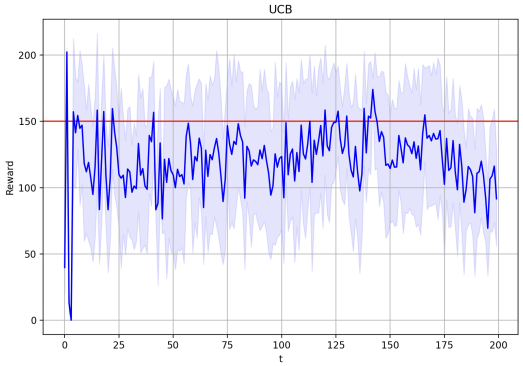

Figure 5: UCB reward

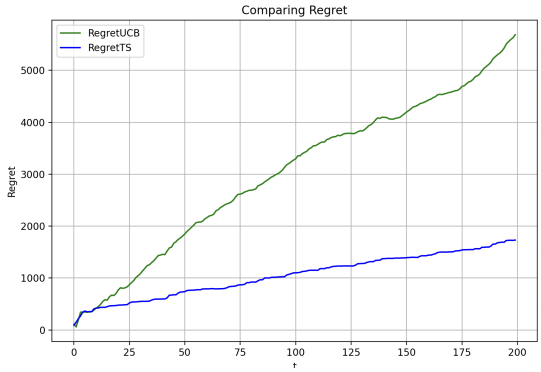

Figure 6: Cumulative regret

The results obtained show that Thompson Sampling converges more quickly to the optimal solution (in the graph, the red line which is the clairvoyant) and in fact the cumulative regret has a smaller slope since Thompson Sampling is based on a priori information and the choice of arm to pull is drawn from a beta distribution representing this knowledge. Regarding the UCB algorithm, it quickly converges to the optimal solution but since the choice of a configuration to pull depends on the bound, on which the computation is based, it may happen that a suboptimal configuration is pulled and in fact the graph is more "unstable." As a result, although the cumulative regret grows almost linearly, the slope of the line is greater than that of the Thompson Sampling.

4 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. The same algorithms as before have been used; the problem faced in this case was how to evaluate **alpha ratios** and **units of product sold** to approach the optimal value: every day, these parameters are updated through the simulation of the user experience, so that they can be used for the next simulation.

4.1 UCB-1

The algorithm follows the same steps as the previous case and they are not reported for the sake of clarity. After selecting the super-arm, simulating the user experience, collecting the reward and updating the algorithm parameters, every day the **alpha ratios** are estimated in this way:

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

alpha_ratios_ucb = estimate_alpha_ratios(old_starts_ucb, total_seen_daily_ucb)
old_starts_ucb += total_seen_daily_ucb
```

where:

- **old_starts:** it is a vector of 6 elements initialized at $\overline{0}$. It represents the number of times each product is seen as the first in the user experience (the first element of the vector corresponds to the possibility of exiting the e-commerce). After every simulation, this vector is updated by summing the new values taken by the result of the simulation of the current day.
- **starts:** it is a vector of 6 elements. It represents the number of times each product is seen as the first in the user experience in the current day (the first element of the vector corresponds to the possibility of exiting the e-commerce).
- **np.sum(new_starts):** it represents the total number of users that visited the e-commerce in the current day.

Moreover, every day the **number of units bought** are estimated as follows:

```
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

for p in range(len(pulled_config_indexes_ucb)):
    total_sold_product_ucb[p, pulled_config_indexes_ucb[p]] += units_sold_ucb[p]
    item_sold_mean_ucb[p][pulled_config_indexes_ucb[p]] =
    ↪  estimate_items_for_each_product(
        item_sold_mean_ucb[p][pulled_config_indexes_ucb[p]],
```

```
total_bought_since_day_before_ucb[p][pulled_config_indexes_ucb[p]],
units_sold_ucb[p],
total_sold_product_ucb[p][pulled_config_indexes_ucb[p]])
```

where:

- **mean:** it represents the last output of the method, which has to be updated with the new parameters.
- **seen_since_day_before:** it is a 5x4 matrix which saves the number of times a product with a certain price has been bought till the day before.
- **unit_sold:** it is a 5-size vector in which each element is the number of times the corresponding product has been bought in the current simulation.
- **total_sold:** it is a 5x4 matrix which saves the number of times a product with a certain price has been bought till the current day.

4.2 TS

The estimations of the **alpha ratios** and the **number of products sold** in the case of Thompson Sampling take place in the same way as the UCB-1 and they are not reported.

4.3 Results

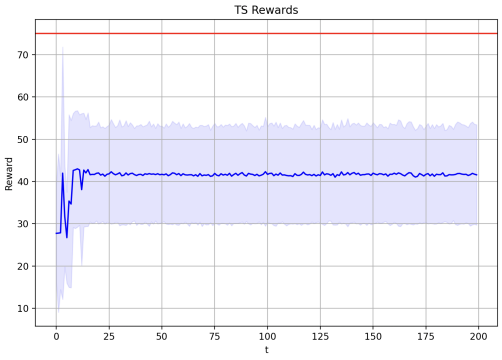

Figure 7: TS reward

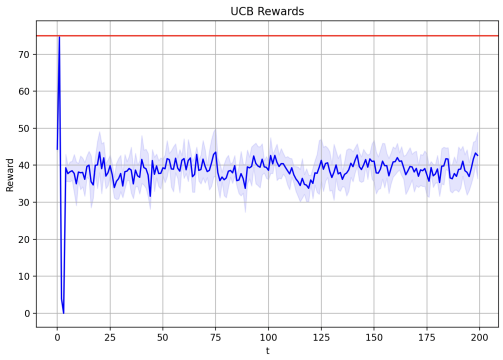

Figure 8: UCB reward

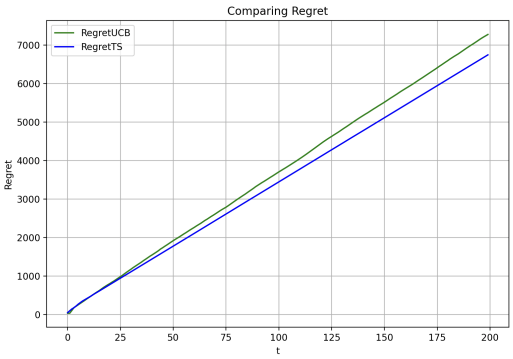

Figure 9: Cumulative regret

For step 4 the results were compared with a clairvoyant that, as seen in the figure, is smaller than in step 3 but still larger than the best solution obtained from the Thompson Sampling and UCB algorithms since, the number of units sold on which the clairvoyant calculation is based, is also a variable to be estimated. The consequence of this gap between best reward and clairvoyant is that the cumulative regrets of both algorithms have very high slopes while always keeping Thompson Sampling lower than UCB. Regarding the reward, the considerations to be made are the same as those made in step 3.

5 Step 5: Optimization with uncertain graph weights

This step reproduces the same conditions as the third step. In this case, unlike the fourth step, together with the conversion rates, also the graph's weights are unknown. This means that these parameters have to be estimated day by day through the simulation of the user's experience on e-commerce. The graph's weights are the probabilities that a customer clicks on a secondary product given the primary product.

Here the environment object is initialized with a different customer class with respect to the previous steps because that customer class is characterized by the parameter *graph_probabilities* which is initialized as a 5x5 matrix with *1* everywhere except for the main diagonal with *0*

```
customer_class_aggregate.graph_probabilities = np.ones((5, 5))
np.fill_diagonal(customer_class_aggregate.graph_probabilities, 0)
```

The way the algorithms work is the same as the previous steps, so it is reported just the additional part aimed at estimating the unknown parameters noted above.

5.1 UCB-1

The *graph_probabilities* are updated every day, through the results taken by the simulation of the customers experience, in the following way:

```
total_bought_day_before = np.copy(total_bought)
total_bought += (units_sold_ucb + units_sold_ts)

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]
```

Where:

- **customer_class.graph_probabilities[product, :]:** it is a row from a 5x5 matrix which represent the product's *graph_probabilities* of the last update.
- **total_bought_day_before[product]:** it is an element of a 5-size vector which keeps track of the number of times each single product has been bought till the day before.
- **customer_class.units_clicked_starting_from_a_primary[product, :]:** it is a row from a 5x5 matrix which keeps track of the number of times a product has been clicked as secondary with a given primary one.
- **total_bought[product]:** it is *total_bought_day_before[product]* plus the number of times the product has been bought in the current day.

5.2 TS

The estimation of the **graph_probabilities** in the case of Thompson Sampling takes place in the same way as the UCB-1 and it is not reported for the sake of brevity.

5.3 Results

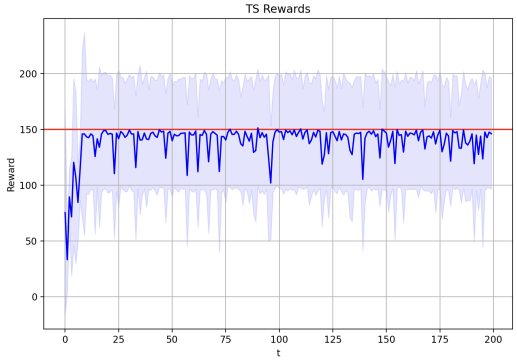

Figure 10: TS reward

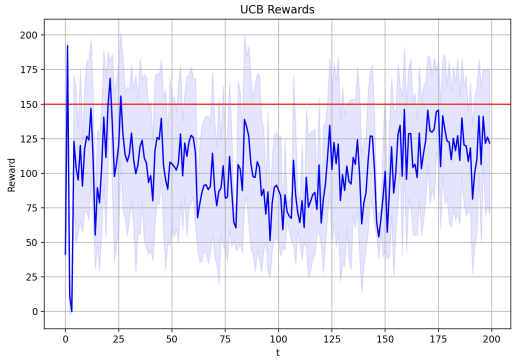

Figure 11: UCB reward

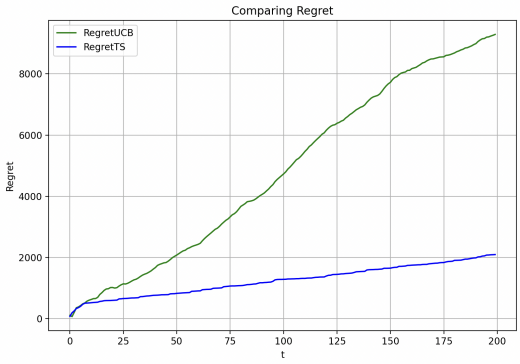

Figure 12: Cumulative regret

The reward considerations for this step are similar to those made previously for the previous steps but both algorithms perform worse (can be seen from the graph of the most "unstable" trends) since there is no longer only one unknown variable, the conversion rates, but also graph probabilities. This introduces a higher probability of choosing a suboptimal arm at each time step. In particular, the reward obtained from the UCB, at half time, is significantly lower than the clairvoyant and this causes an increase in the slope of the cumulative regret which eventually returns with the initial incline because the reward stabilizes.

6 Step 6: Non-stationary demand curve

Now we face the problem where the demand curves could be affected by some abrupt changes. In our case, the simulator presents 4 possible abrupt changes correlated with the variation of the *reservation_prices*. The switch between one *reservation_prices_configuration* and another has been reported as a switch among different phases. Two different approaches of the same algorithm have been used to sort out the same problem:

- *UCB-like approach with change detection*
- *UCB-like approach with sliding windows*

Here, for the first time, the *NonStationaryEnvironment* is initialized. The only two differences with respect to the common environment are: it keeps track of the time step; the methods *simulation* and *graph_search* we have seen so far, are substituted with other two methods which take into account the possible change of the phase as it is written above.

Then the classes of the two versions of the algorithm are initialized as the common UCB, each of them acts similarly with respect to the classic version, indeed the actions of pulling the arm and updating their parameters are almost the same as the ones seen in step 3. The main differences are due to the fact that in this case, we have to detect whether the end of the window of days or any change in the conversion rates and then, in both cases, to reset the parameters of the algorithms: the *mean* matrix is reset to 0 whereas the *upper confidence bound* matrix is reset to infinite.

Here the *clairvoyant* value is computed in a different way. It always represents the best obtainable reward and it is used to make a comparison with the new results, but this time, what is obtained through the *evaluate_abrupt_changes_clairvoyant* method is a 4-size vector which contains a clairvoyant value for each phase.

```
def evaluate_abrupt_changes_clairvoyant(configurations, max_units_sold,
↪  reservation_prices, n_users, n_phases):
max_reward = np.zeros(n_phases)
for phase in range(n_phases):
    for config in configurations:
        opt_reward = 0.
        for product in range(5):
            if reservation_prices[phase][product] >= config[product]:
                # evaluate the reward for
                opt_reward += max_units_sold * n_users * config[product]
        opt_reward = opt_reward / n_users
        if opt_reward > max_reward[phase]:
            max_reward[phase] = opt_reward
return max_reward
```

6.1 SW_UCB

The idea behind this algorithm is that, since the demand curves can be subject to change, old samples can lose importance with time and they can affect negatively the result of the experiment.

Following this approach, it has been chosen to have a sliding window of 25 days: it means that the *reservation_prices_configuration* is updated every 25 days but, since the fact that a single experiment is characterized by a specific number of days, the last phase will last more than the others if the *number_of_days* is higher than 100. Whenever the phase changes also all the parameters which are used for the updating of the *SW_UCB* are set to 0:

```
if t != 0 and t % window == 0:
    if sw_ucb_phase < 3:
        sw_ucb_phase += 1
    total_seen_ucb = np.zeros((n_products, n_prices))
    total_seen_product_per_window_ucb = np.zeros(n_products)
```

where:

- *total_seen_product_per_window_ucb* keeps track of the number of users which have seen the single product within the current window of 25 days.

6.2 CD_UCB

On the other hand, at the end of every day, the *CD_UCB* algorithm computes, after the update of its parameters, the conversion rate for each pair product-price:

```
self.conversion_rates[product][pulled_config[product]].append(bought[product]/seen[product,
↪  pulled_config[product]])
```

where:

- *bought[product]:* it is the number of times a product has been bought in a day.
- *seen[product, pulled_config[product]]:* it is the number of times a product has been seen with a certain price in a day.

And then it checks if there has been a change. The way it does is the following:

```
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
```

The result of the method *detect_change* is used for whether changing the phase or not. As we can see in the snippet, a parameter *size* and a threshold δ have been set. The first one is used to compute first, the mean over the old conversion rates as *last_mean* and then, the mean over the latest ones as *new_mean*. This happens for each pair product-price and, whenever one among those two ratios is greater than or equal to δ then a change is detected in the conversion rates. Therefore the phase is updated and finally the parameters are reset.

```
if cd_ucb_learner.detect_change(pulled_config_indexes_cducb):
    if cd_ucb_phase < 3:
        cd_ucb_phase += 1
    total_seen_cducb = np.zeros((n_products, n_prices))
    total_seen_product_cducb = np.zeros(n_products)
```

6.3 Results

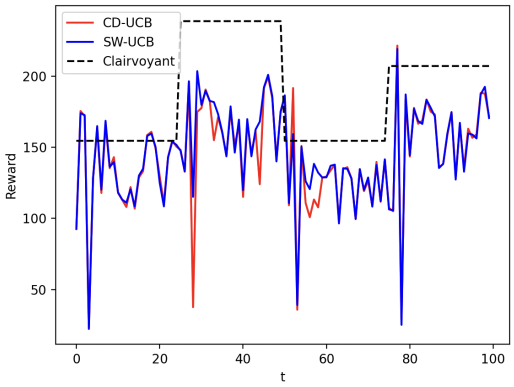

Figure 13: CD-UCB and SW-UCB reward

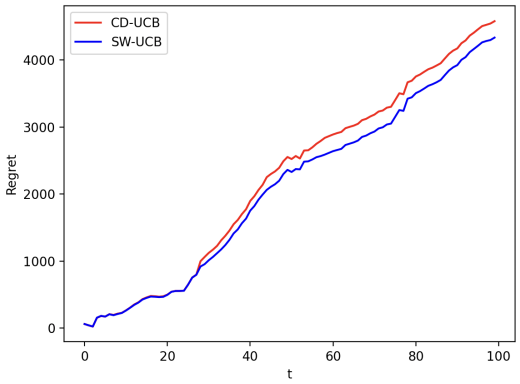

Figure 14: Cumulative regret

Both algorithms, Sliding Window UCB and Change Detection UCB, have similar performance since changing the window leads to a corresponding change in reward that occurs almost simultaneously in both cases. Furthermore, both algorithms approach clairvoyant although the CD-UCB is slightly worse in the second window. The window changes are clearly visible in the cumulative regret plot, particularly with regard to the transition between the first and second windows.

7 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.

The learners used in this step are the same used in step 4. The only difference is that, in place of the standard *Environment* class, the *ContextualEnvironment* class is used: it is exactly the same as the original one but it is initialized with no customer class and it has an extra method which is the setter of the customer class:

```
def select_costumer_class(self, customer_class):
    self.customer_class = customer_class
```

It will be updated as the Context is updated. The learners' algorithms remain unchanged.
Here the *clairvoyant* represents the best obtainable reward and it is used to make a comparison with the new results, but this time, what is obtained through the *evaluate_contextual_clairvoyant* method is a 2-size vector which contains a clairvoyant value for each "one feature" customer class.

```
    def evaluate_contextual_clairvoyant(configurations, customer_classes,
    ↪  prices):
    rewards_per_customer_class = []
    for customer_class in customer_classes:

    ↪  rewards_per_customer_class.append(evaluate_rewards_per_combination(configurations,
    ↪  customer_class, prices))
    return rewards_per_customer_class
```

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:

$$p_l \mu_l + p_r \mu_r \geq \mu \tag{3}$$

where μ, μ_l and μ_r are the expected rewards, p_l and p_r are the probability weights and the lower bound μ is:

$$\mu = \bar{x} - \sqrt{-\frac{\log(10e - 10)}{2|Z|}} \tag{4}$$

where $\tilde{x}$ is the expected reward computed by simulating the learner and $|Z|$ is the size of the data. In the *Python* code a method that evaluates this condition is implemented in a support class for step 7 called *ContextClass*:

```
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

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

def assign_prob_context_occur(self, time):
    return 1 / (2 ** (time / 14)) * 100
```

Since the way the context generation algorithm works, i.e., by using a greedy approach, the optimal solution is not guaranteed and it might get stuck in a sub-optimal branch of the splitting tree, leading to a sub-optimal solution.

7.1 Results

7.1.1 Results of costumer class 1

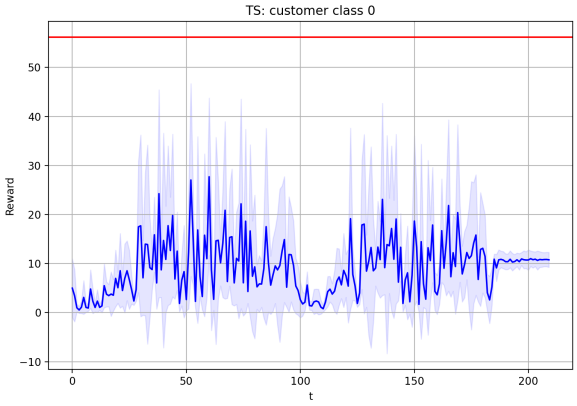

Figure 15: TS reward

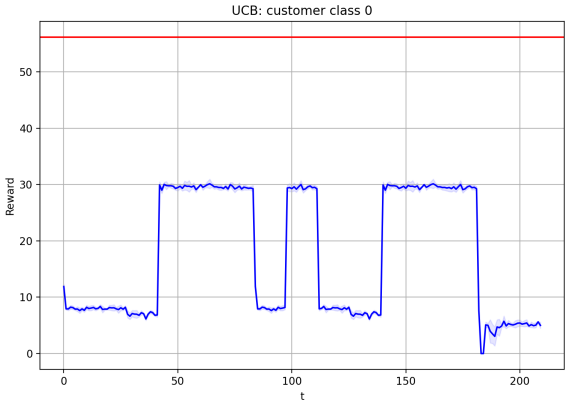

Figure 16: UCB reward

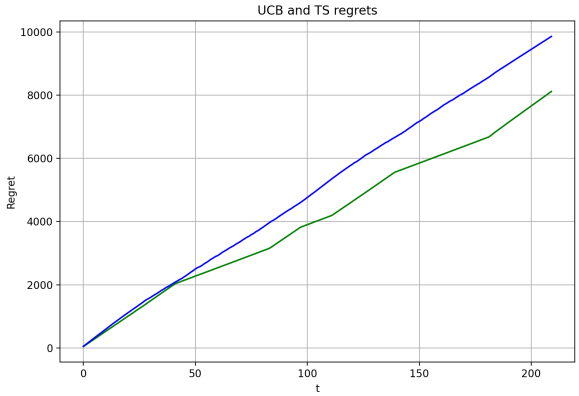

Figure 17: Cumulative regret

7.1.2 Results of costumer class 2

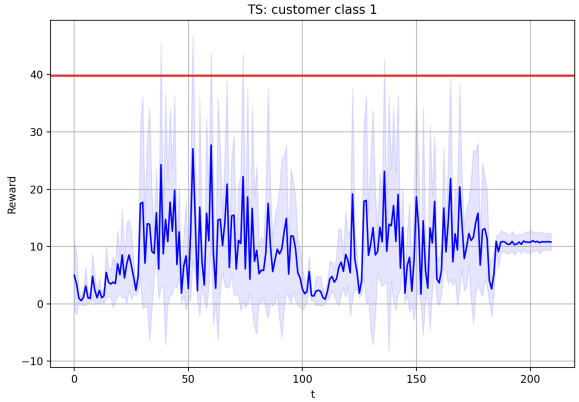

Figure 18: TS reward

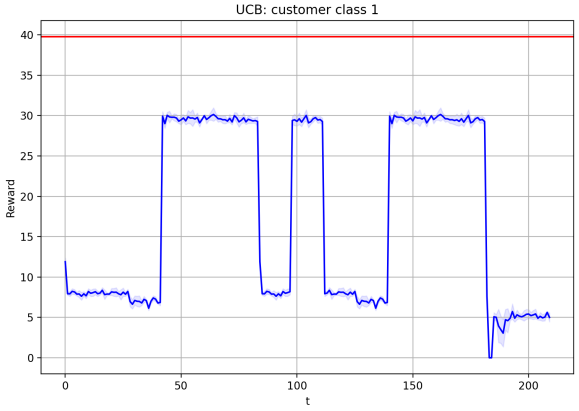

Figure 19: UCB reward

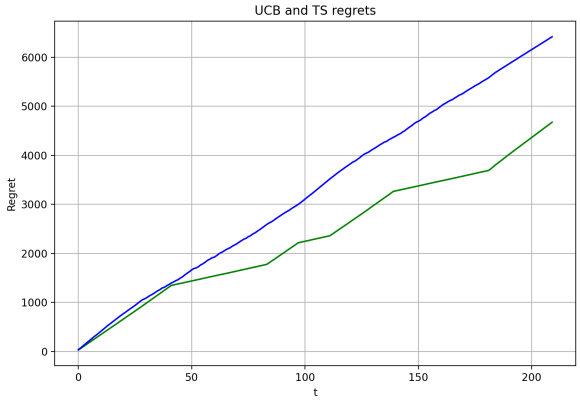

Figure 20: Cumulative regret

7.1.3 Results of costumer class 3

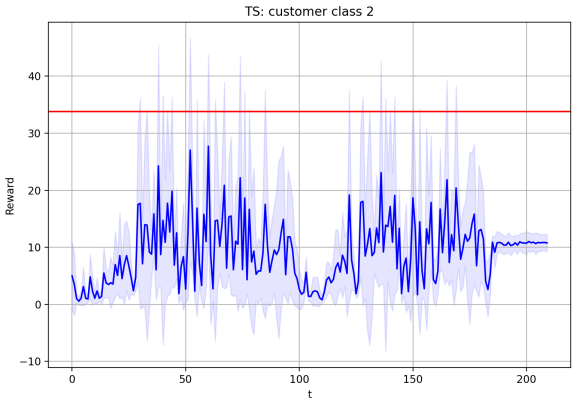

Figure 21: TS reward

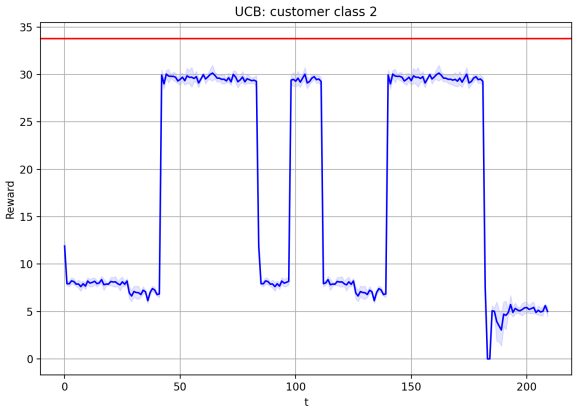

Figure 22: UCB reward

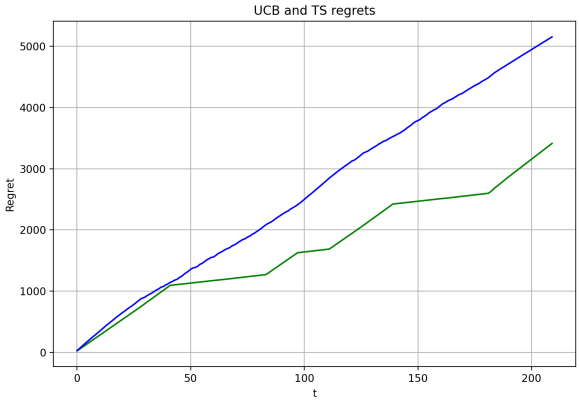

Figure 23: Cumulative regret

The results obtained in the final phase of the project show the rewards and the regrets collected for each customer class: they are evaluated by using the clairvoyant values of the corresponding context combined with the collected rewards. Since the context-generation algorithm applies a greedy policy each time a split occurs, the obtained rewards might not correspond to the actual optimal solution; nonetheless, the Thompson Sampling algorithm settles on a (sub-)optimal solution at the end of the simulation, as it is possible to see on the plots related to the rewards of each customer class, while the UCB-like algorithm is very sensible to splitting context, as we can see on the plots of the collected rewards. However, the variance of UCB is much smaller than the variance of TS. Another remark, the cumulative regret plots (green line: UCB, blue line TS) show how the cumulative regret evolves as context change: this is particularly visible for UCB, while the TS has a smoother trend due to the higher variance of the rewards.