Project of Online Learning Applications,
Nicola Gatti

Online Learning Applications, Pricing and Social Influence

- Attorre Francesco 10618456
- Ceruti Giacomo 10603486
- Repole Giampiero 10543357
- Zamponi Ettore 10808192

Contents

1 Introduction

1.1 Analysis

Imagine an ecommerce website which can sell an unlimited number of units of 5 different items without any storage cost. In every webpage, a single product, called primary, is displayed together with its price. The user can add a number of units of this product to the cart. After the product has been added to the cart, two products, called secondary, are recommended. When displaying the secondary products, the price is hidden. Furthermore, the products are recommended in two slots, one above the other, thus providing more importance to the product displayed in the slot above. If the user clicks on a secondary product, a new tab on the browser is opened and, in the loaded webpage, the clicked product is displayed as primary together with its price. At the end of the visit over the ecommerce website, the user buys the products added to the cart.

Firstly we have to analyze the content of the pages:

- Primary Product, price and buy button
- Secondary Product slots
  First secondary product
  Second secondary product

The secondary product are shown only if the user buy at least 1 unit of the primary product, they link to a new page where the clicked product is the current primary. The second slot of the two secondary products has a lower visibility than the first one.

1.1.1 Real World Scenario

For a more precise analysis we identified the website as a stationery e-commerce. Where the product sold are:

- Notebook
- Pen
- Pencil case
- Diary
- Calculator

1.2 Constraint

- **Reservation price**: the user buy x items only if the price is under the user's reservation price
- **Product visibility**: if a product is already been visited by the user, that product will never be clicked again by the same user
- **Graph Probability**: the probability that the user click on a secondary product is dependent from the primary product
- **Numbers of unit sold**: The number of items a user will buy is a random variable independent of any other variable

1.3 Product distributions

1.3.1 Prices

The price of every primary product is a variable to optimize. Assume that there are four values of price for every product and that the price can be changed once a day.

```
"products": [
  {
    "name": "notebook",
    "cost": 1,
    "price": [1.99, 2.49, 2.99, 3.49]
  },
  {
    "name": "pen",
    "cost": 0.5,
    "price": [0.99, 1.20, 1.5, 1.99]
  },
  {
    "name": "pencil case",
    "cost": 2,
    "price": [3.99, 4.5, 4.99, 5.99]
  },
  {
    "name": "diary",
    "cost": 3,
    "price": [8.99, 9.99, 11.99, 14.99]
  },
  {
    "name": "calculator",
    "cost": 4,
    "price": [5.99, 6.99, 7.99, 9.99]
  }
]
```

We have set the prices of every product in increasing order.

1.3.2 Price index

Since we had 4 different prices per product, we use an array that select the index of the price to use in every of the 5 products.

1.3.3 Margins

Every price is associated to a known margin.

We set a fixed cost for every product, and we evaluate the margin of gain for every product as the difference between the current price selected and the cost of the product. For example if we consider the first configuration of prices:

```
prices = [1.99, 0.99, 3.99, 8.99, 5.99]
margins = [0.99, 0.49, 1.99, 5.99, 1.99]
```

1.3.4 Secondary

For every primary product, the pair and the order of the secondary products to display is fixed by the business unit and cannot be controlled.

We realize the relative array in this way:

```
secondary = [
    [1, 3],
    [0, 2],
    [3, 0],
    [2, 4],
    [0, 1]
]
```

Where the rows are associated to the primary products, and the columns represent the secondary products. The first column is the first secondary product, and the second the second one. We always have to check that the secondary association does not contain the primary product itself.
The correlation between the primary and secondary products are:

- **Notebook**: Pen, Diary
- **Pen**: Notebook, Pencil case
- **Pencil case**: Diary, Notebook
- **Diary**: Pencil case, Calculator
- **Calculator**: Notebook, Pen

1.4 Users distributions

1.4.1 Binary Features

We had to distribute 3 different classes of users over 2 binary features.

```
"features": ["student", "professor"]
```

The classes that we analyze are:

- Student

  ["true", "false"]
- Worker

  ["false", "false"]
- Professor

  ["false", "true"]

We develop different parameters for each class of users, that potentially differ for:

- Demand curves of the 5 products
- Number of daily users
- α ratios
- Number of products sold
- Graph probabilities.

1.4.2 Users' correlations with products

We identified different trends for each user class.

- **Student**:

  The number of students is larger than the other classes.

  Their preferred products are: *Notebook, Pen*.

  Economically bounded, so they have lower conversion rates than the other classes.

  If they purchase, the number of items bought is higher than the other classes as they need more items.

- **Worker**:

  The worker class is less interested in the products sold, but they trend to buy more expensive items.

  Their preferred products are: *Pen, Diary*.

  Lower graph probabilities, since they are not really interested on the type of products.

- **Professor**:

  Has common interests with Student but they purchase a lower number of items

  Their preferred products are: *Pen, Diary, Notebook*.

1.4.3 Numbers of users and α ratios

Every day, there is a random number of potential new customers (returning customers are not considered here).

They customers on the website are seen as visitors. We consider 2 parameters per class: **mean** and **standard** deviation of a *Normal distribution*. Every user that lands on our website, can land on the webpage in which one of the 5 products is primary or on the webpage of a product sold by a (non-strategic) competitor.
Every day we have a new vector of ratios for each user class:

- α ratios are realizations of independent **Dirichlet random variables.**
- Call α_i the ratio of customers landing on the webpage in which product P_i is primary, and call α_0 the ratio of customers landing on the webpage of a competitor.
- The ratios are a fraction of the total number of users.

For the α ratios we use a different set of weights per class, as input of a Dirichlet distribution. An example of weights is written below:

```
      "alpha_weights": [0.4, 0.3, 0.3, 0.4, 0.6, 0.4]
```

1.4.4 Graph probability

Every secondary product below a primary one has a probability that the user click on it, these probabilities depends on:

- Purchase probability of the primary product.

- Probability to observe the slot in which the secondary product is displayed, and the click probability of the secondary product is conditioned to the purchase of the primary.

We consider that the user click on the product only if he has purchase the primary, and we multiply the probability of clicking over the second secondary product with a *lambda* parameter:

```
    "lambda": 0.8
```

From the text we had to consider two different scenario:

- The graph (vector of ratio) is fully connected and therefore all the edges have strictly positive probabilities.
- The graph (vector of ratio) is not fully connected and therefore some edges have zero probability.

We distribute the graph probability with two parameters for each user class:

```
    "min_probability": 0.6,
    "max_probability": 1
```

Then we use an **Uniform** distribution with these parameters for creating the graph of probabilities.
For a not fully connected graph, randomly we set some probabilities to 0:

```
    graph_weights_not_fully_connected = np.copy(graph_weights)
    for z in range(len(classes_idx)):
        for i in range(numbers_of_products):
            how_many_set_to_zero = int(round((numbers_of_products-2) *
                npr.random()))
            set_to_zero = (npr.choice(numbers_of_products, how_many_set_to_zero,
                replace=False)).astype(int)

            for j in set_to_zero:
                j = int(j)
                graph_weights_not_fully_connected[z, i, j] = 0
```

1.4.5 Conversion rates

For every product, the conversion probability associated with each price value is a random variable whose mean is unknown.

They are distributed with a **Uniform** with min and max values different for each classes:

```
    conv_rates = np.zeros((len(classes_idx), numbers_of_products,
        different_value_of_prices))
    for i in range(len(classes_idx)):
        min_demand = classes[classes_idx[i]]["demand"]["min_demand"]
        max_demand = classes[classes_idx[i]]["demand"]["max_demand"]
        conv_rates[i] = npr.uniform(min_demand, max_demand, (numbers_of_products,
            different_value_of_prices))
```

They represent the probability of purchase at least one item of a product.

1.4.6 N items bought

The number of items a user will buy is a random variable independent of any other variable; that is, the user decides first whether to buy or not the products and, subsequently, in the case of a purchase, the number of units to buy.

```
n_items_bought = np.zeros((len(classes_idx), numbers_of_products))
for i in range(len(classes_idx)):
    max_item_bought =
        classes[classes_idx[i]]["n_items_buyed"]["max_item_bought"]
    n_items_bought[i] = npr.uniform(1, max_item_bought,
        numbers_of_products).astype(int)
```

1.5 Environment & Simulator

1.5.1 Environment

Firstly we had to consider how to create the environment and how the logic flow works, for Scalability reasons we decide to write a configuration file with all the simulator, products and users data.

```
"simulator": {
    "days": 10,
    "max_item_bought": 15,
    "max_users": 300,
    "seed": 2000,
    "secondary": [[2, 3],[0, 4],[1, 3],[1, 2],[3, 0]]
},
"product": {
    "products": [...],
    "different_value_of_prices": 4,
    "numbers_of_products": 5,
    "lambda": 0.8
},
"users": {
    "features": ["student", "professor"],
    "classes": [
      {"name": "Student",...},
      {"name": "Worker",...},
      {"name": "Professor",...}
    ],
    "graph_weight_fully_connected": 1
}
```

Simulator

- **days**: default numbers of days of the simulation.
- **max_items_bought**: plotting limit.
- **max_users**: plotting limit.
- **seed**: fixed for Numpy.
- **secondary**: already explained in the previous chapter.

Products

- **products** list of products with all it's value (already explained in the previous chapter).
- **different_value_of_prices**: fixed from the text.
- **numbers_of_products**: fixed from the text.

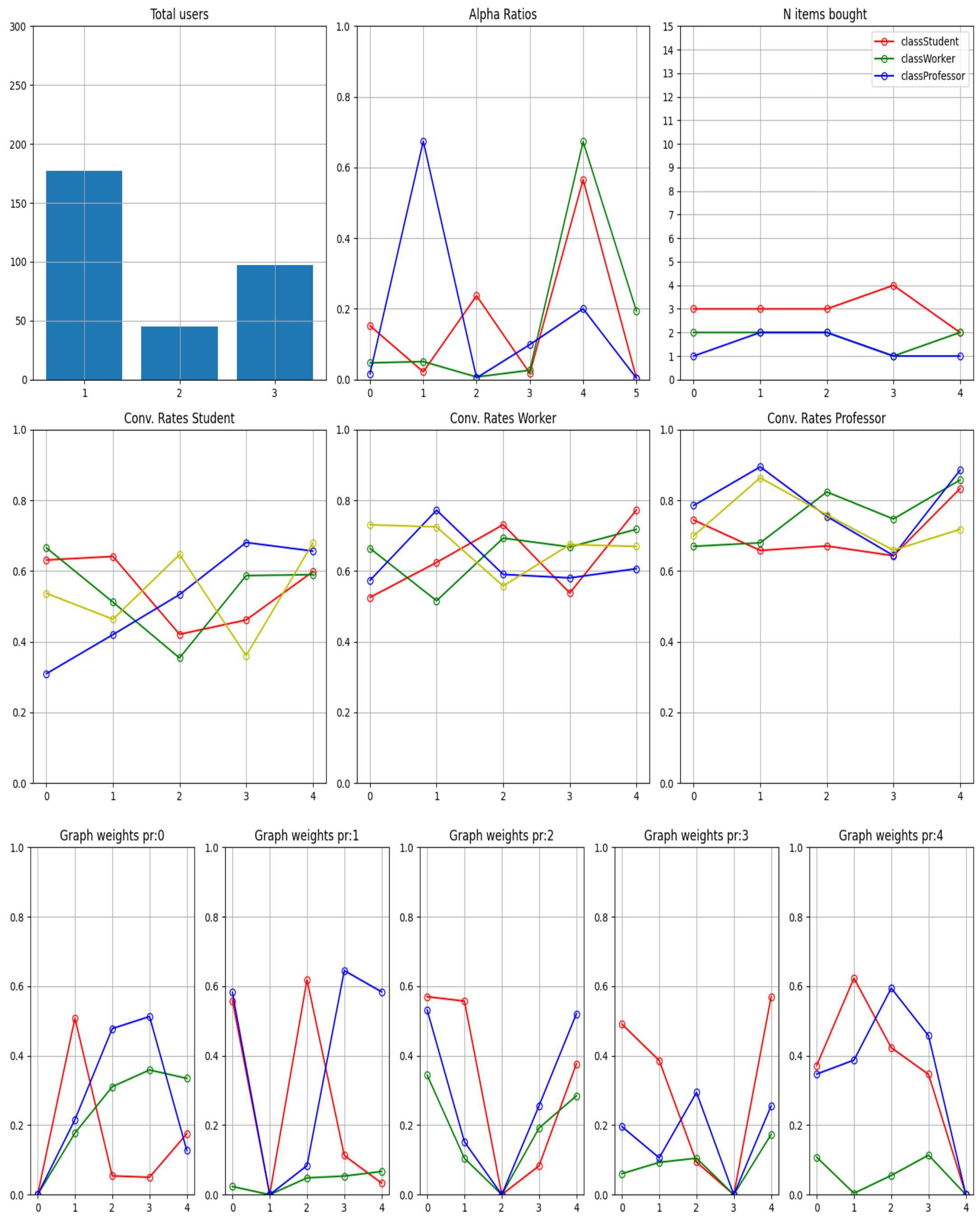
- **lambda**: defined to 0.8.

Users

- **features**: the list of binary feature names.
- **classes**: list of users distributions parameters already explained in the previous chapter.
- **graph_weight_fully_connected**: since the graph can be fully connected or not, with this parameter we choose to use a fully connected graph or not.

1.5.2 Users distributions

In the following plot, we can see the reward for the different distributions over the users.



1.6 Simulator

1.6.1 website_simulation

This methods iterate the simulation over the list of user classes passed as parameter on the simulator, it works by using the alpha ratios over the *total_users* parameter of the *user* class. This is enough to simulate the website's profit margin for a single day, but if we want to simulate multiple days, for each day we update:

- *total_user*
- α *ratios*
- *simulator.margins*
- *simulator.prices*

and using the Environment variable *days* we iterate over *website_distribution* for that number of times.

1.6.2 Simulator

The simulator works by firstly loading all the product's distributions, then the user's one. The effective simulation works with 10 as the default number of iterations, and with:

- Primary product
- User class selected

The user class contain all the users distributions, a method for obtain the number of items to buy differently from the user class selected:

```
def get_n_items_to_buy(self, product):
    npr.seed(data["simulator"]["seed"])
    return int(npr.uniform(1, self.n_items_bought[product]))
```

The website simulation module takes care of calling the simulator for each user class and iterate over the products. It considers the alpha ratios on the total number of users of the selected class. The number of iteration is computed as product of the numbers of users and the alpha ratios:

```
alpha = [0.15237489, 0.02211131, 0.2373585 , 0.01769267, 0.56572612,
    0.00473651]
n_users = 177
(alpha * n_users).astype(int) => [ 26, 3, 42, 3, 100,  0]
```

The alpha ratios are composed by 6 elements, the first one represents the competitor webpage and the others are related to the 5 products.

The simulator is divided into 2 main steps:

1. **Primary product**: It defines the rewards as array of zeros, and it evaluates the conversion factor and the relative reward:

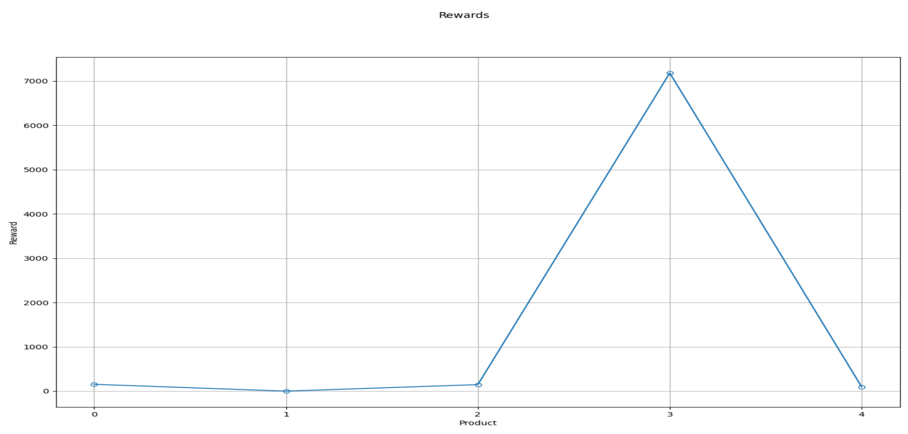
```
conversion_factor =
    bernoulli.rvs(user_class.conv_rates[j][self.prices_index[j]], size=1)
rewards[j] = self.margins[j] * \
            user_class.get_n_items_to_buy(j) * \
            conversion_factor
```

The conversion factor is given from a *bernoulli* distribution, where the conversion rates of the user class relative to the product and price considered are the probability of purchasing the product. The reward instead is computed as the product of: Profit margin for the price selected, the number of items that the user bought, (that at every iteration is evaluated as an *uniform* distribution between 1 and the maximum number of items bought by the class) and obviously the conversion factor previously calculated.
Then it adds the product to the current visited_primaries modifying the graph probabilities to 0 for it. If the conversion_factor is not False, proceed. In the other case, return an array of Zeros as the array of rewards for the simulation.

2. **Secondary product**: It retrieves the secondary products, with another *bernoulli* distribution over the **graph_probabilities** for the selected product, and chooses if call recursively the simulation over them.
   At the end we retrieve the summary reward for the single user. Of course the probability in input to the second secondary product is multiplied for the lambda factor, for reducing the observable probability of that slot.

```
        first_secondary = int(self.secondary_product[j][0]) # Observation vector
            for every product!
        if bernoulli.rvs(arr[first_secondary], size=1):
            rewards += self.simulation(first_secondary, user_class)
```

In the following plot, we can see the reward for the different products:



2 Optimization algorithm

2.1 Introduction

We aim to optimize the cumulative expected margin over all the products. We have developed a greedy algorithm with these instruction:

- At start, initialize every item with the lowest price.
- Evaluate the marginal increase of the reward received, by increasing the price of a single product by a single level.
- Choose the price configuration that has the highest marginal increase respect to the the 5 products.
- Iterate on finding a best configuration, until any new configuration of prices isn't better than the actual configuration.

2.2 Load the data

Firstly we need to load the users and the simulator distribution. We select the lowest price configuration and retrieve the reward obtained.

```
    self.prices_index = [0, 0, 0, 0, 0]
    self.reward = website_simulation(self.sim, self.users)
```

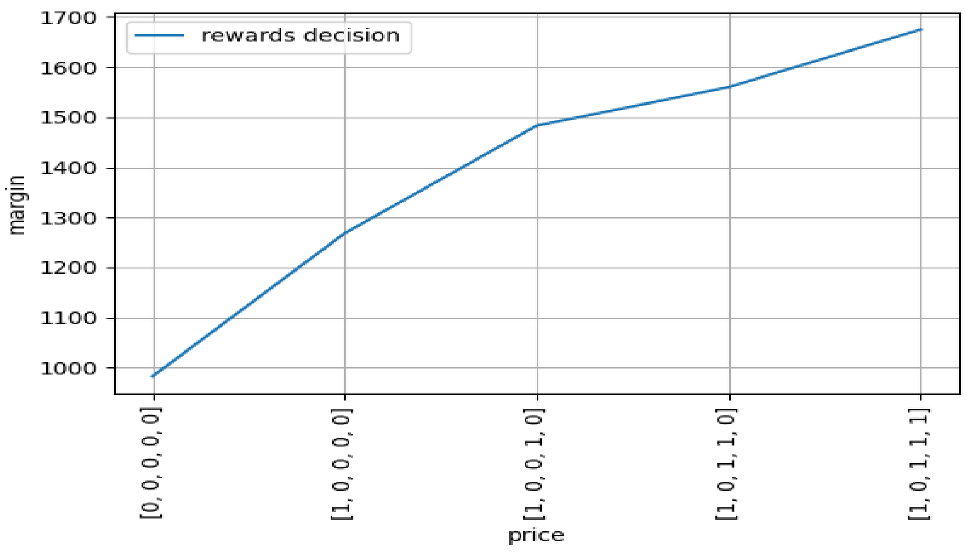
And we create a list of variables for logging the search process of the algorithm.

2.3 Algorithm

The algorithm keep a temporary index that is the current one, and iterate on the 5 products by increasing the prices chooses if isn't the maximum one yet. on every iteration we retrieve the reward by choosing that index of prices, and confront it with the maximum one. At the end of the iteration we choose the maximum one (if is not the previous configuration) and we call the same function recursively.

2.4 Result

We can see the evolution of the algorithm by the price pulled with the highest reward.



3 Optimization with uncertain conversion rates

3.1 Introduction

In the next chapters the following setting is analyzed : every users behaviour on our website is observed in a specific iteration (this is a simulation of one day of website activity) with a different (daily) price setting for each product.
The objective is to find the optimal price setting to have a maximum observed reward from sells. In order to achieve this goal two Multi Armed Bandit algorithm are used, UCB and TS.
The observations obtained from website sells are used to update these algorithms decisions parameters, which are used to select the next day candidate arm, how this is done is the core of the algorithm and its based upon a reward expectation (by simulation) given the observable and estimated parameters. Specifically in the step number 3 the probability that an user buy an item once he visit the product primary page is unknown and needs to be estimated, instead step 4 and 5 deals with other unknown parameters which can still be approximated with observations.

3.2 UCB

Upper Confidence Bound, in its standard version is guaranteed to minimize the regret. By having in mind how this algorithm works the specific implementation for our problem setting will be described. The most important difference is the non [0,1] reward retrieved from website, which makes:

$estimate\ conv\ rate \quad ...$

$confidence\,bound\,for\,product\,i\,on\,price\,j\ := c_{ij}$

$expected\,reward\,for\,product\,i\,on\,price\,j\ := \bar{x}_{ij};$ (this refers to a [0,1] scaling of the actual rewarded value)

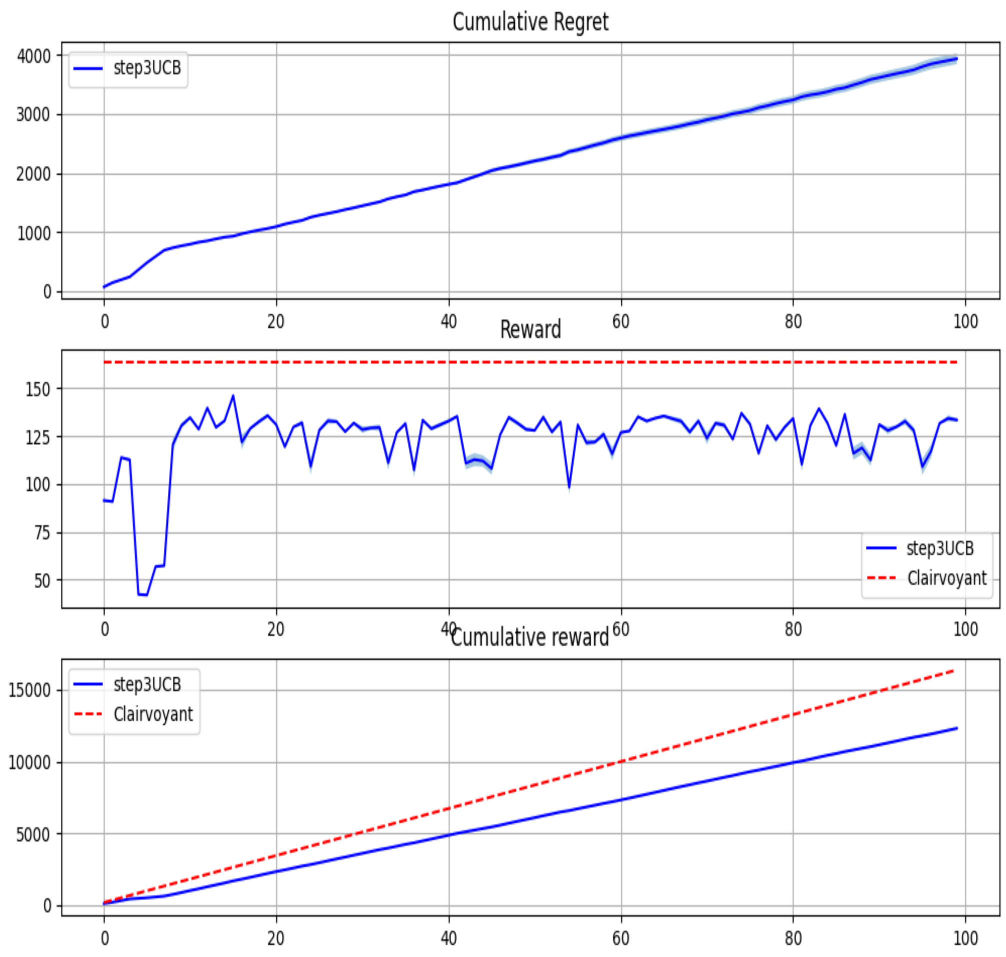
```
        self.prices_index = [0, 0, 0, 0, 0]
        self.reward = website_simulation(self.sim, self.users)
```

The algorithm work following these main steps:

1. Fix the mean probability to 0 and the bound length of every price to +infinite;
2. Compute the probability mean (m) of a product visited to be sold at a certain price;
3. Compute the bound length (w) of each price for each product;
4. Pull the arm with the highest reward m + w.

3.2.1 Result

After a small number of iterations, the UCB find the best combination of arm, even if some days it could pull others arms to try new combination with an big bound length.
A fast convergence is traduced in a small regret, in fact after 8/9 iteration the algorithm founds the best combination and the regret's grade decreases.

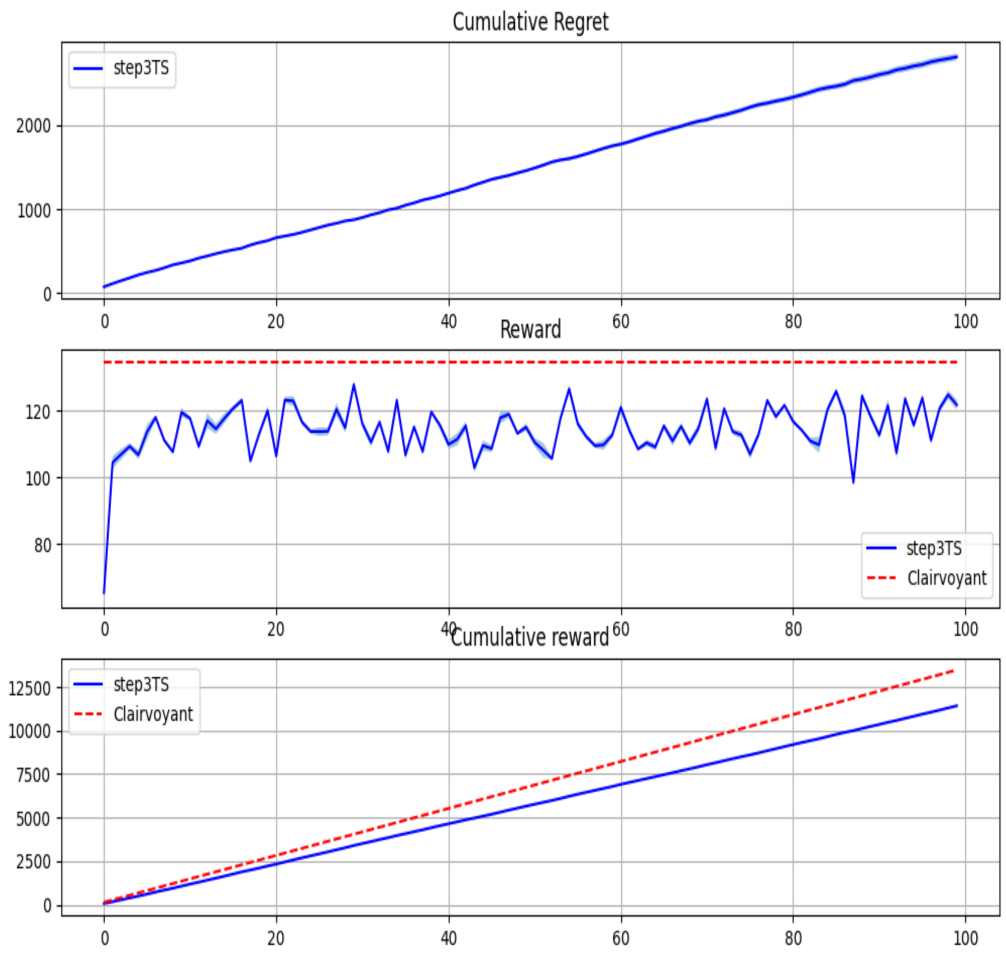


3.3 Thompson Sampling

Thompson Sampling is a stochastic algorithm, instead of computing the sum of the mean value and the bound length for each arm, it samples a value for every pair of product and price, based on a probability distribution and choose the highest sample per product.

3.3.1 Result

In this scenario the TS converges pretty fast near to the optimal combination, indeed the regret is not so big. Also the clairvoyant cumulative reward is so similar.



3.4 Comparison

As the graphs shown, both the algorithms perform well in this case, the only difference is that the UCB is deterministic meanwhile the TS is stochastic. The TS has a lower cumulative regret because it find faster the best arm to pull, instead of doing exploration like the UCB when an arm is not ever be pulled, TS keep the best arm also after the first iteration.
This way of acting reward the TS in a static situation, instead of trying some new arm to explore that could be potentially better in the current solution.

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

4.1 Introduction

In this Step, these are the uncertain:

- conversion rates

  Updated with the mean of the observed ones.

- α rates

  Observed as the ratios of the first products visited.

- number of items sold per product

  Observed as the mean as the observed ones of the purchases.

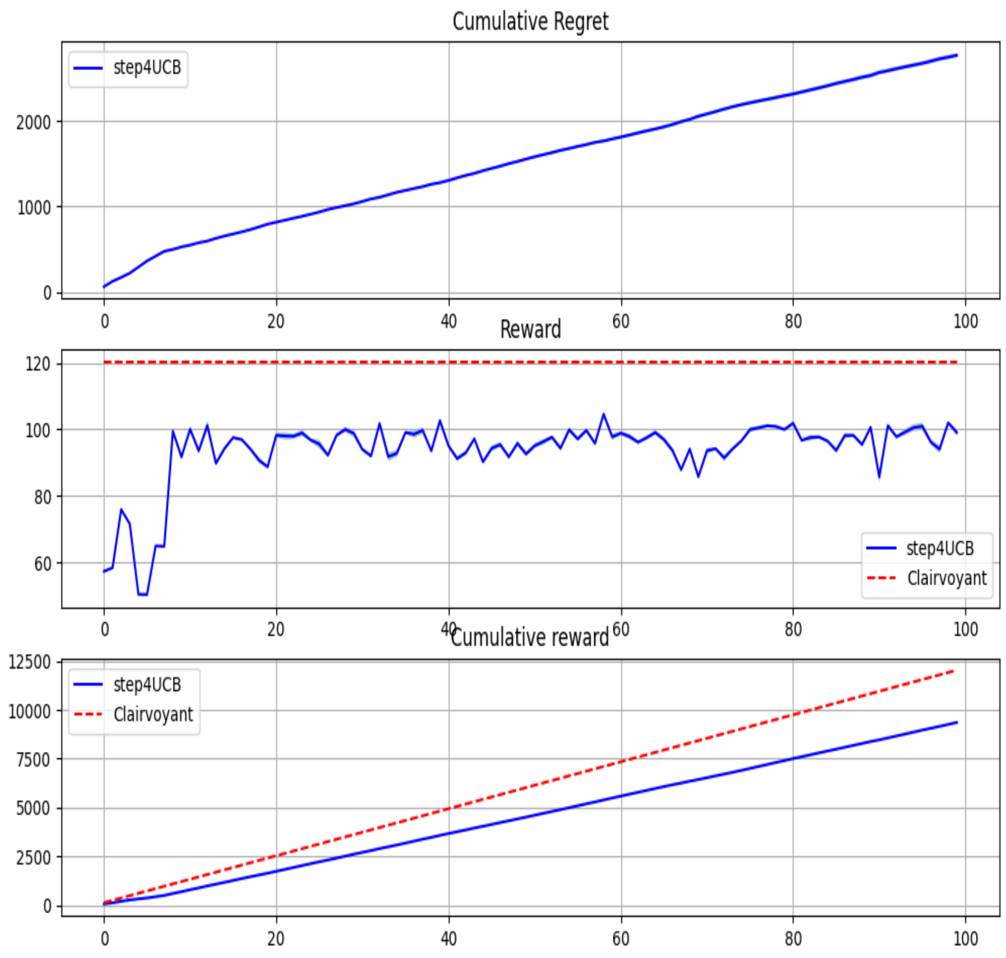
So the Learner cannot estimate on these parameters and it just observes them, since the learner can update it's parameters with the website visible information's of the customer.

4.2 Results

Due to the loss of information respect to the step 3, we can observe some changes in the performance of our algorithms:

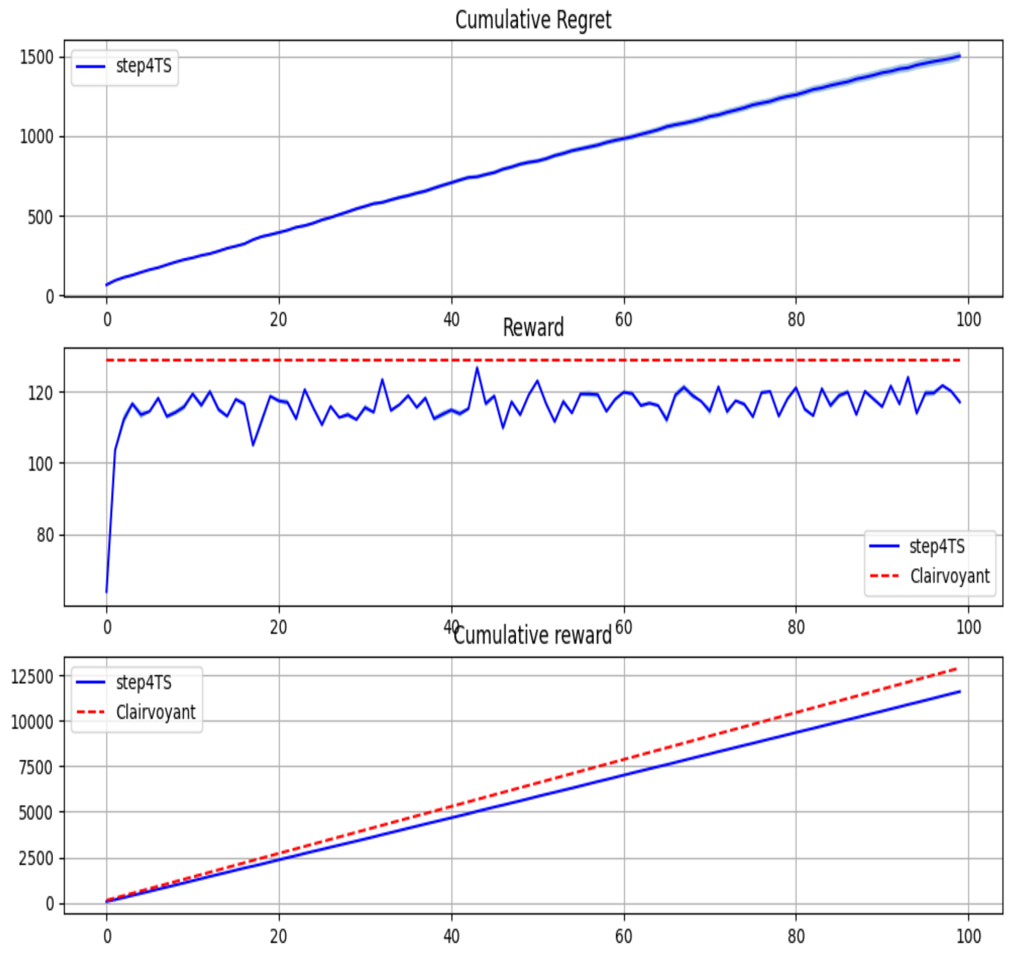
4.2.1 UCB

This version of the UCB has less fluctuations while converging to an optimal value meanwhile it is the wrong one respect to the clairvoyant reward. It happens because there are more random unknown parameters in our simulation so also the optimal reward changes.



4.2.2 TS

By estimating the previously described parameter, our algorithm can go faster to the converged valued, and has a low standard deviation on the reward function. The cumulative regrets instead is pretty linear



5 Optimization with uncertain graph weights

5.1 Introduction

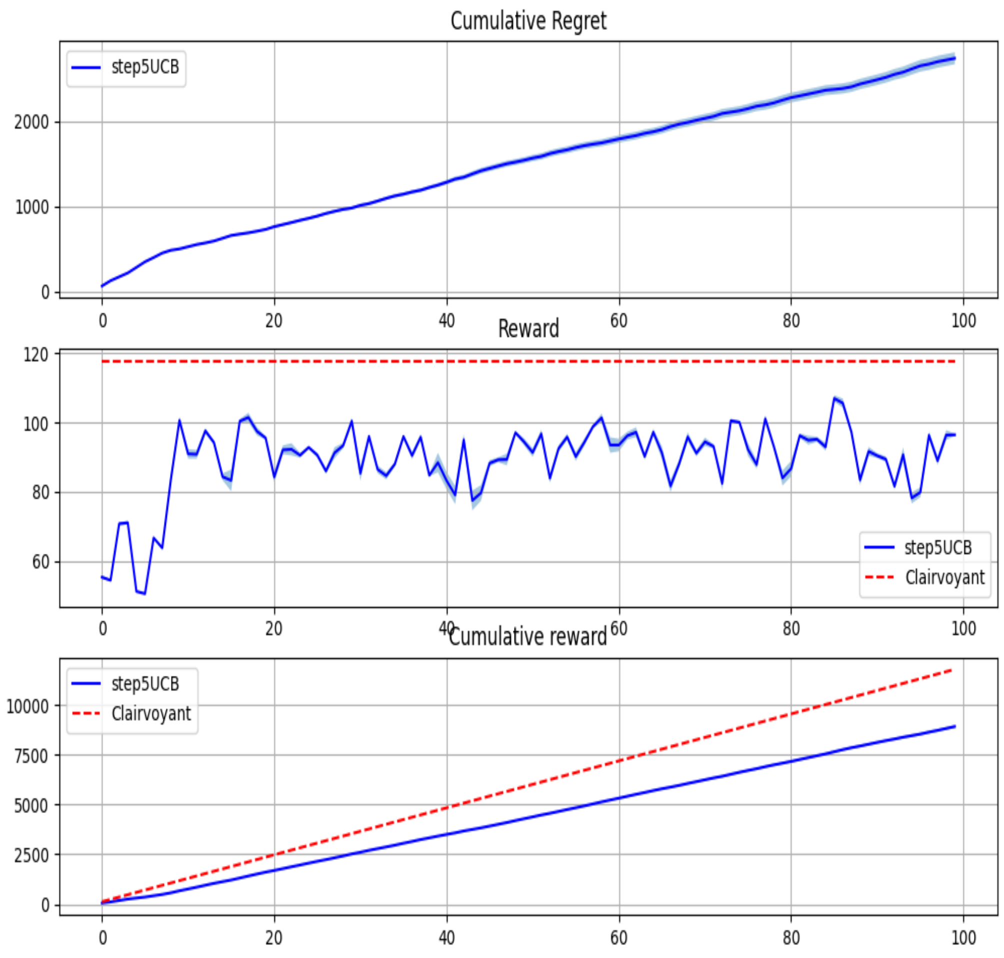
In this Step, the unknown parameters are just the graph weights, we try to learn them by the list of visited products, over the simulation's iterations.

5.2 Results

We have two different Clairvoyant rewards between the algorithms, because we choose the best iteration values to generate two better plots. With the graph probabilities unknown we could not reach an upper reward than before

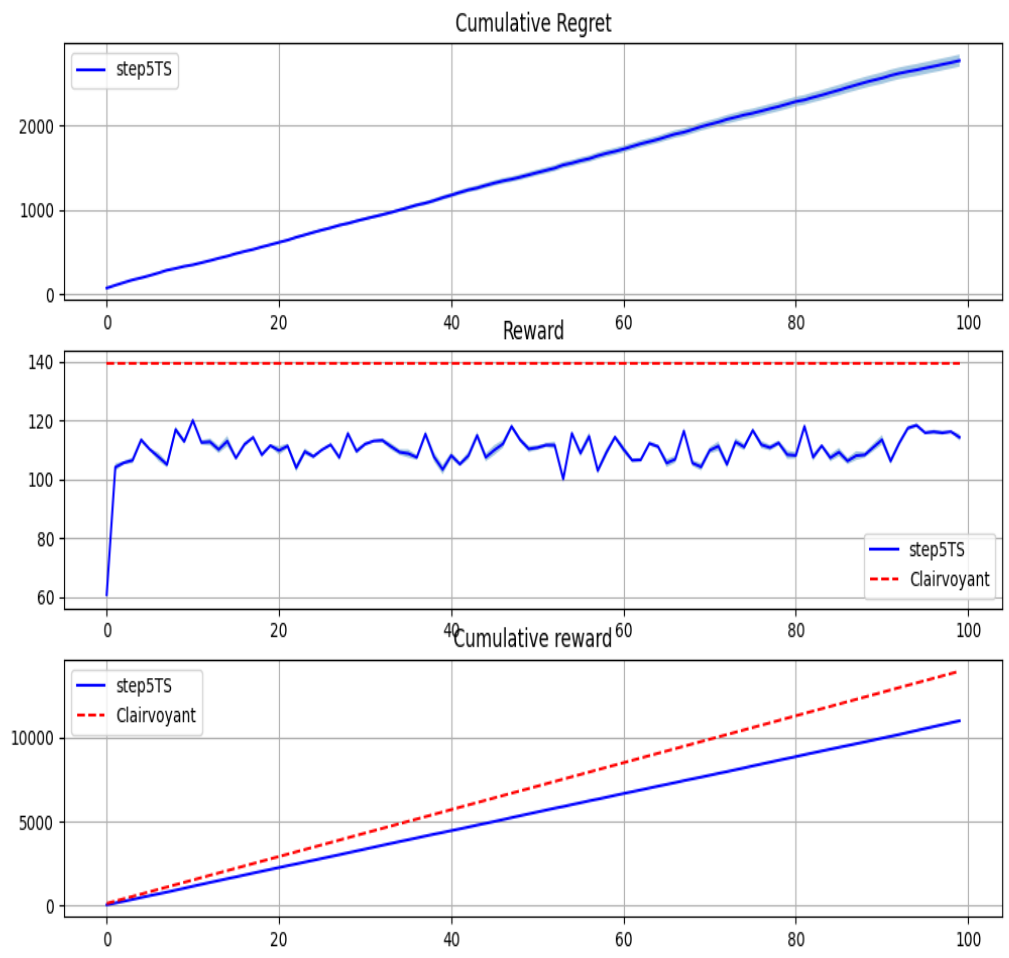
5.2.1 UCB

The reward starts low but UCB learns the local maximum after having estimated the best graph values, after 10-15 iterations, then it starts to converge with an high standard deviation.



5.2.2 TS

The Cumulative reward is linear, in the regret instead a lower standard deviation over the 20 iterations is reached, where the learner has estimated the optimal graph weights,



6 Non-stationary demand curve

6.1 Introduction

In this scenario it is supposed that the demand curve could be subjected to abrupt changes, generating a new completely unexpected setting.
The *UCB* algorithm and the *Sliding-Window UCB* algorithm have to face this situation, detect the sudden change and evaluate the performance in the new unexpected situation.

6.2 UCB Algorithm

In this situation the UCB can detect the abrupt change in the demand curve considering every arm per product. We expect a difference distribution of the cumulative reward moving away from its previous mean.

6.2.1 Result

The simulation considered 100 iterations and 50 daily iteration by the user where the abrupt change happens two times, at 50 and at 80 iterations.
There are two distinct change of direction in the following graphs, corresponding to the abrupt change. The reward suddenly decreases and the increase trying to learn with the new demand curve.

6.3 Sliding Window - UCB

The *Sliding Window UCB* works exactly as the UCB only with the fact that it realize the means and widths of the bound in a different way, that is taking into consideration only the last τ previous day. So, in this way, the sliding window version forgot all the others samples of the time horizon (T number of iteration) except for the last τ samples.

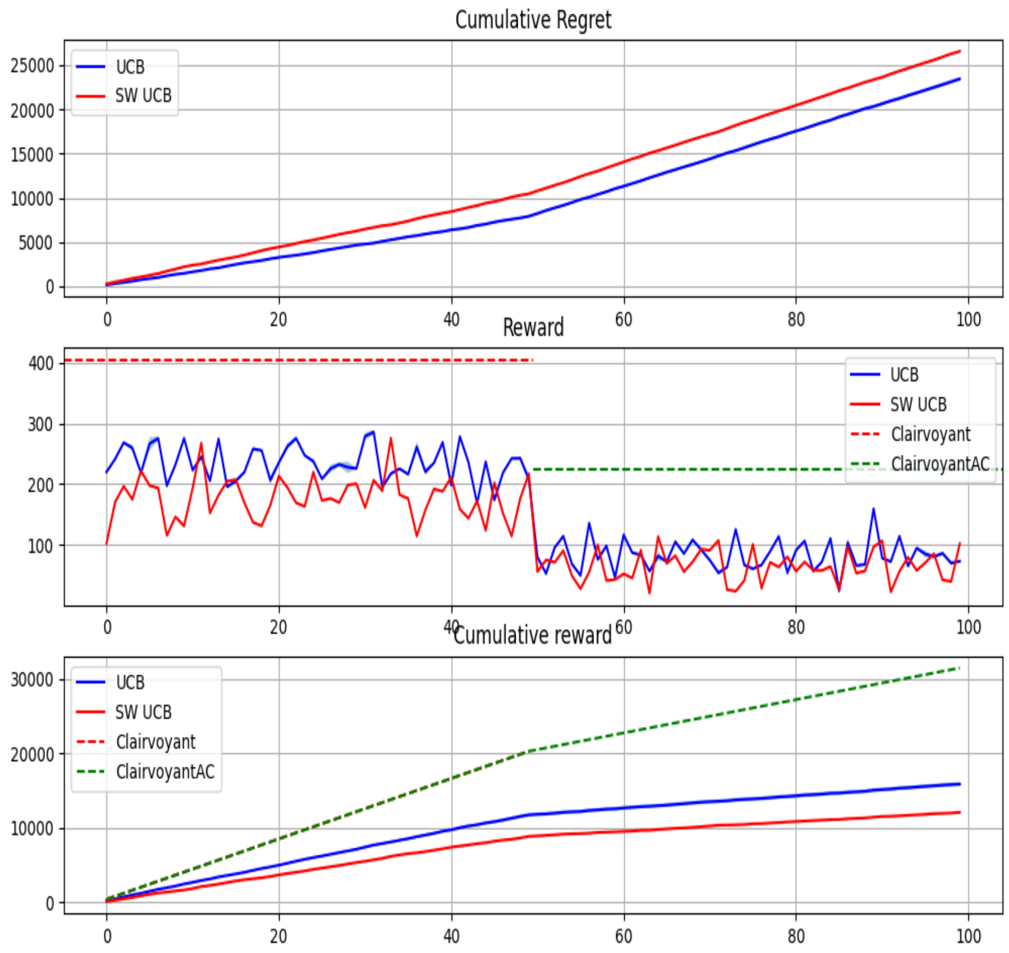
$$\tau \propto \sqrt{T}$$

6.3.1 Result

The SW-UCB acts so similar to the UCB, having changes of curve distribution close to the moment in which happen the abrupt changes.

6.4 Comparison

Even if the UCB and SW graphs are so similar, the UCB performs slightly better, the reward of the UCB is higher always and indeed the regret of the UCB is lower than that of Sliding Window UCB.
Theoretically the UCB adapt itself only after a bit of iteration rather than the SW that requires τ iterations to remove the old samples.


Cumulative Regret
UCB
SW UCB
25000
20000
15000
10000
5000
0
0
20
40
60
80
100
Reward
400
300
200
100
UCB
SW UCB
Clairvoyant
ClairvoyantAC
0
20
40
60
80
100
Cumulative reward
30000
20000
10000
0
UCB
SW UCB
Clairvoyant
ClairvoyantAC
0
20
40
60
80
100

7 Context generation

7.1 Introduction

Here we have the same scenario as the step 4 (optimization with uncertain conversion rates, alpha ratios, and number of items sold per product). It's possible to generate a new context every two weeks. Each context should be optimized independently, and the split happens simultaneously on all the contexts. With context we are talking about considering every feature of the 3 user classes , (Student, Worker, Professor) and finding the best configuration value where the reward is optimized.

7.2 Implementation

7.2.1 Context Algorithm

The feature's context is not known to the learner. A context is a subspace of the feature values where all the learners are updated independently, all on a relative sub feature space. Every time we need to generate a new context we find the best learner considering the lower confidence bound on all the learners, using the average of the older rewards.

With the context generation over 2 weeks, the algorithm cannot reach the clairvoyant reward.

Initialization of the context:

```
users_classes_to_import = [0, 1, 2]
best_prices_per_class = [[3, 3, 3, 1, 3], [3, 3, 2, 3, 2], [3, 3, 3, 3, 3]]
best_not_aggregated_reward = find_not_aggregated_reward(best_prices_per_class, env)
context_learner = ContextualLearner(features=features, n_arms=env.n_arms,
    n_products=numbers_of_products)
if UCB_LEARNER:
    root_learner = UCBLearner(users_classes_to_import, different_value_of_prices)
else:
    root_learner = TSLearner(users_classes_to_import, different_value_of_prices,
        DAYS)

root_node = ContextNode(features=features, base_learner=root_learner)
context_learner.update_context_tree(root_node)
context_generator = ContextGenerator(features=features,
                                     contextual_learner=context_learner,
                                     users_classes_to_import=users_classes_to_import,
                                     days_to_simulate=DAYS,
                                     confidence=0.1,
                                     iteration=SIMULATION_ITERATIONS)
```

Firstly we find the best dis-aggregated prices, considering all the classes of users. Then the context learner is initialized with the feature space, the context node is the chosen algorithm updated later in the tree. For the generation of the context we used the imported classes, the confidence bound, and the last iterations of the day.

iteration of the day

```
        for i in range(DAYS):
    print("Iteration " + str(k) + ": day " + str(i))

    if i % 14 == 0 and i != 0:
        context_generator.context_generation()
    review = np.zeros(numbers_of_products)
    pulled_arms = np.zeros(numbers_of_products)

    for j in range(SIMULATION_ITERATIONS):
        ...
        ...

    learner.update(pulled_arms.copy(), review, visited_products,
        num_bought_products)
    context_generator.update_average_rewards(current_features=name_features)
```

Every 14 days, we generate a new context, choosing the best learner based on their older average rewards, and maybe change the current feature space. Generating newer contexts, we train offline more new learners with these stored parameters, ordering them by lowest confidence bound on the expected rewards. Then we update the learner, and the context with the following parameters retrieved from the simulation:

- pulled arms
- review
- visited products
- number of products bought

The following code is instead about the iterations of the simulator:

```
        for j in range(SIMULATION_ITERATIONS):
            current_features, name_features =
                get_right_user_class(users_classes_to_import)
            learner =
                context_learner.get_learner_by_context(current_features=name_features)

            pulled_arms = learner.act()
            rew, visited_products, num_bought_products, num_primary =
                env.round(pulled_arms, current_features)
            learner.updateHistory(rew, pulled_arms, visited_products,
                num_bought_products, num_primary)
            context_generator.collect_daily_data(pulled_arms=pulled_arms.copy(),
                                                 coll_rewards=rew,
                                                 visited_products=visited_products,
                                                 num_bought_products=num_bought_products[0],
                                                 num_primaries=num_primary,
                                                 features=name_features)
```

Every simulation, we randomly select the combination of feature values to analyze retrieving the learner, then we act on it choosing the best arm to play. We simulate the environment considering the current features, and update both the learner and the context with the retrieved data.

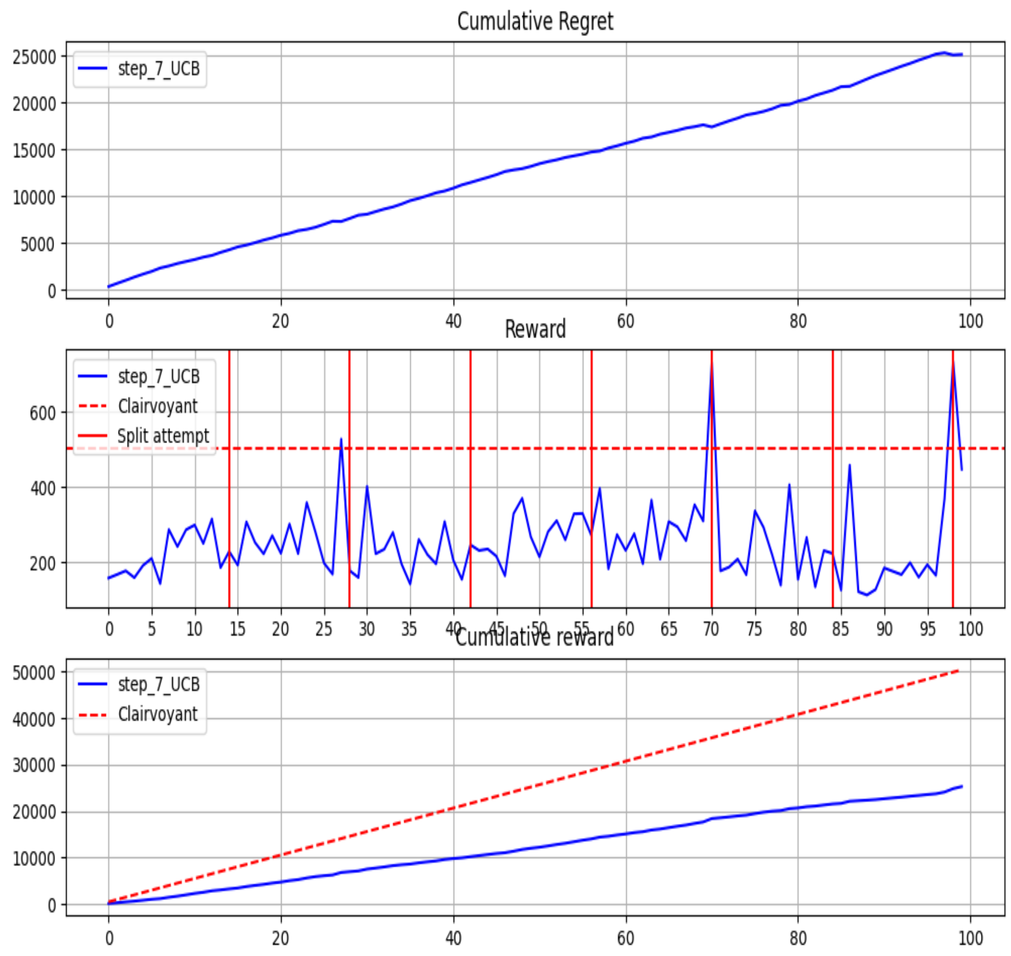
In the following results we can visualize how every 14 days we have a red vertical line, that indicates that probably a newer context has been selected

7.3 Result

The cumulative reward and regret are almost linear, the Reward instead fail to converge, we can see how every 14 days the algorithm is at a fixed mean and then start to learn an upper reward arm before changing context again, it seems that the range of 14 days is too much shorter for let the algorithm converge.

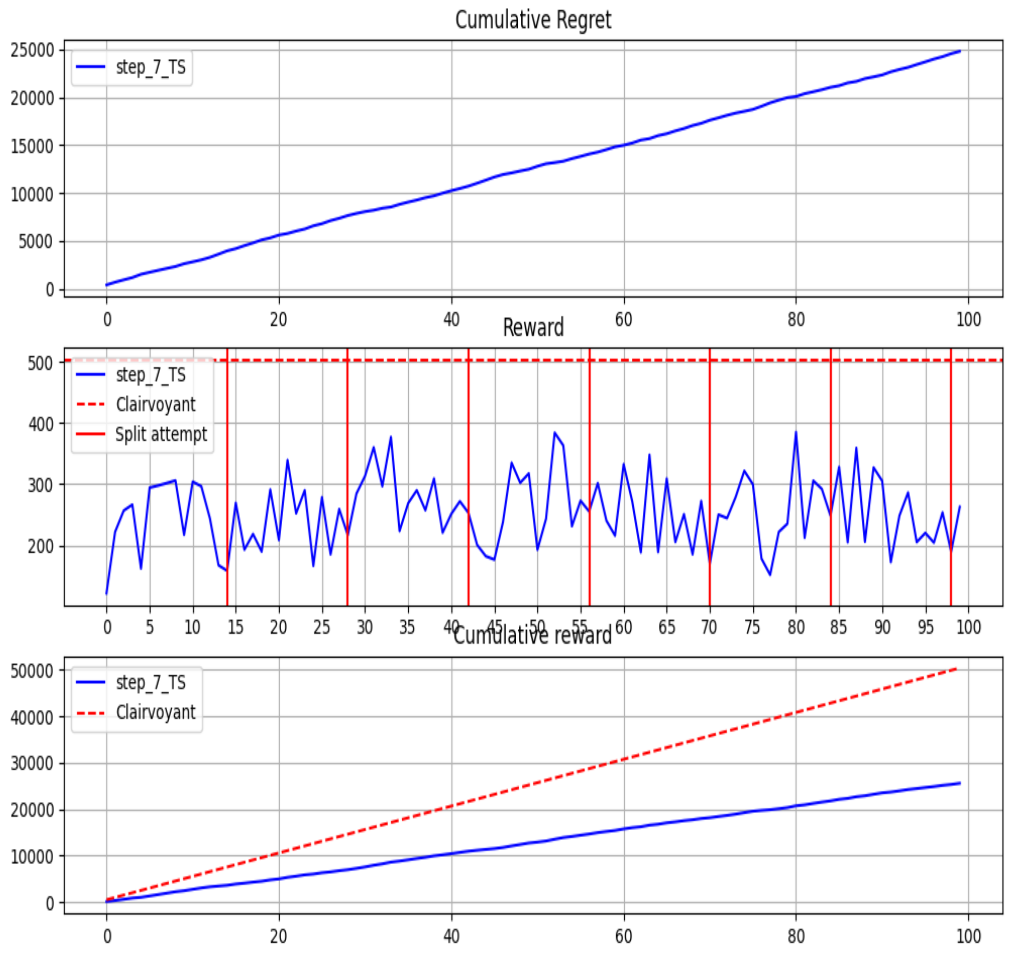
7.3.1 UCB

Here the changing feature brings the learner to continue to pull arms with lower values, then by constantly pulling higher arms, we find higher rewards.



7.3.2 TS

In this version of TS, we fix the conversion rate to a maximum value until an arm has not be pulled yet. The followed graph try to converge to a lower reward considering the clairvoyant. In this case it perform worse than the UCB right here.


Cumulative Regret
step_7_TS
Reward
step_7_TS
Clairvoyant
Split attempt
Cumulative reward
step_7_TS
Clairvoyant