POLITECNICO
MILANO 1863

# PRICING AND ADVERTISING PROJECT

**Pietro Gadaleta (10801034)**
**Adrián Pérez I Alvarez (10913548)**

**Online Learning Applications**
**2022-2023**

POLITECNICO
MILANO 1863

# TABLE OF CONTENTS

# STEP 0: MOTIVATIONS AND ENVIRONMENT DESIGN

*Imagine and motivate a realistic application fitting with the scenario above. Describe all the parameters needed to build the simulator.*

## Motivations

Our company is a Travel Agency, and we sell special travel packages to visit different parts of the world. We want to introduce to our products a new package to visit Milan, which includes flights, activities, hotels, etc.

For this new package, we want to study the best price which will lead to the best benefit. Also, the best marketing strategy to maximize our sales.

## Environment design

The characteristics of our product are the following:

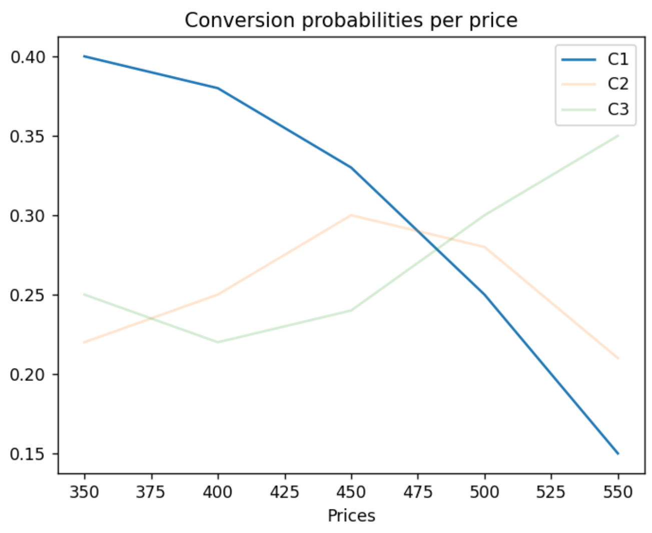
- **Cost for the company**: 210€
- **Possible price values**: [350, 400, 450, 500, 550].

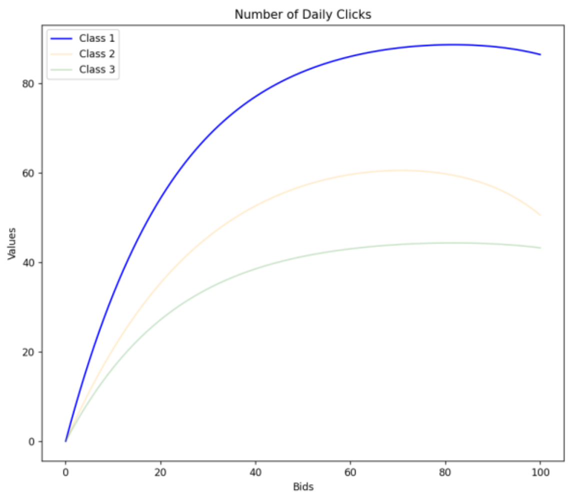
We can observe two binary features through the advertising platform:

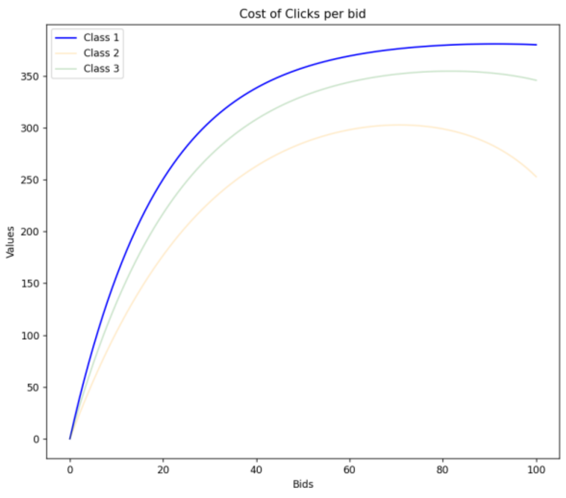
- **F1: Age of the customer** (under or below 45 years old).
- **F2: Country of residence of the customer** (Italy residence or not).

From these binary features, we can cluster our target consumers into 3 different groups:

- **C1**: This are our young potential customers. As young people, their travel style is more conservative and looks for the cheapest options. For this reason, the conversion probabilities decrease as the price increases, reaching a maximum probability on the lowest price. Also, they are very familiar with internet commerce, so they have the biggest daily clicks on our web side. The cost of the clicks is also high for this reason.
    - **F1**: young people
    - **F2**: Foreign or Italian residence (indifferent).
    - **Conversion probabilities**: [0.4, 0.38, 0.33, 0.25, 0.15]
    - **Daily clicks per bids curve**: 100 * (1.0 - np.exp(-4 * x + 2 * x ** 3))
    - **Cost of click per bids curve**: 400 * (1.0 - np.exp(-5 * x + 2 * x ** 3))



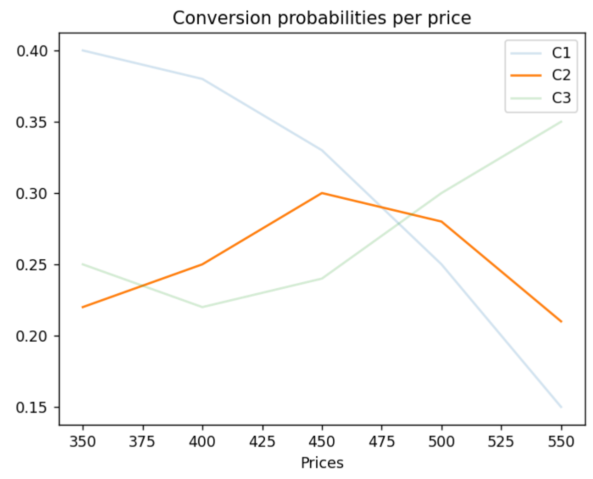


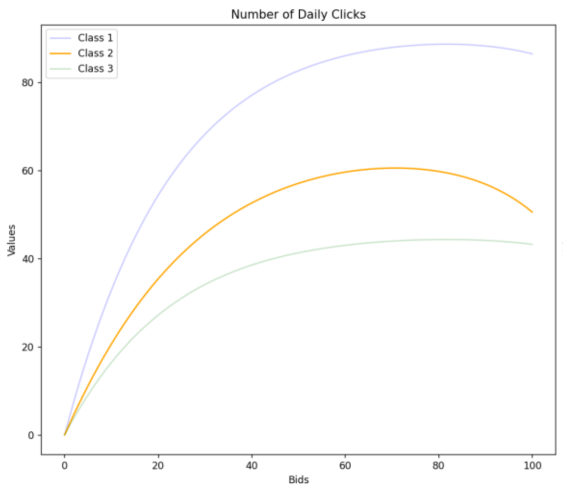


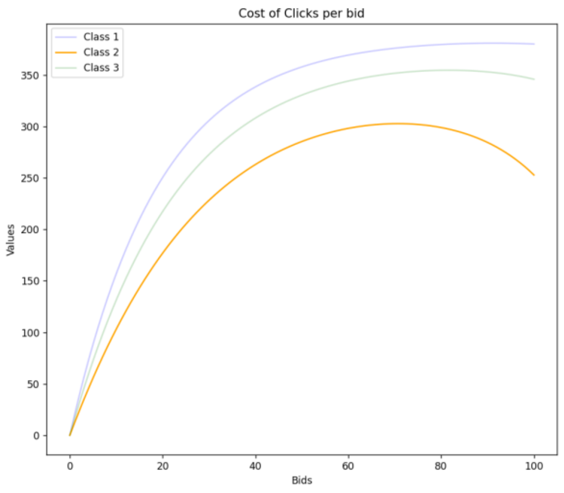
- **C2**: This class corresponds to old people that live in Italy: they are willingness to pay more to have the best accommodation and activities. But, as they already live in Italy, is not so exclusive experience, so they are looking for a good price while keeping a good quality of the product (the best conversion probability occurs in an average price).

  They aren't familiar with internet commerce, so the daily clicks is lower than the young people. The cost of the clicks is also lower.

  - **F1**: old people
  - **F2**: Italian residence
  - **Conversion probabilities**: [0.22, 0.25, 0.30, 0.28, 0.21]
  - **Daily clicks per bids curve**: 80 * (1.0 - np.exp(-3 * x + 2 * x ** 3))
  - **Cost of click per bids curve**: 400 * (1.0 - np.exp(-3 * x + 2 * x ** 3))



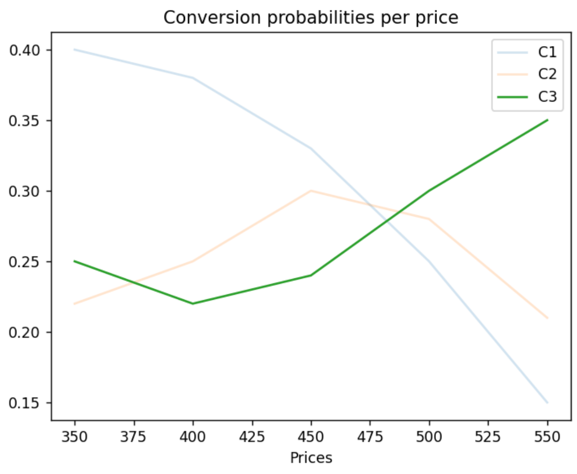


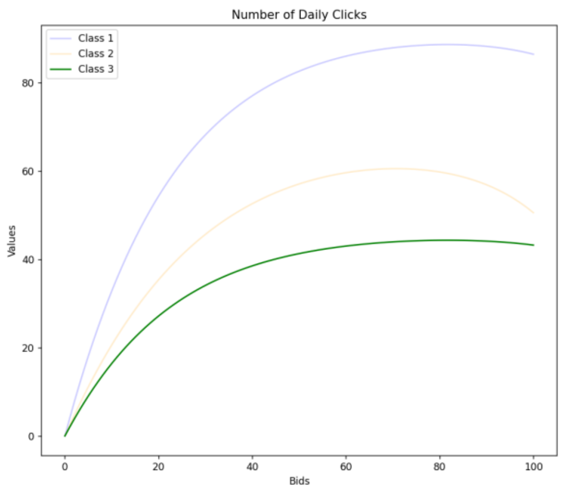


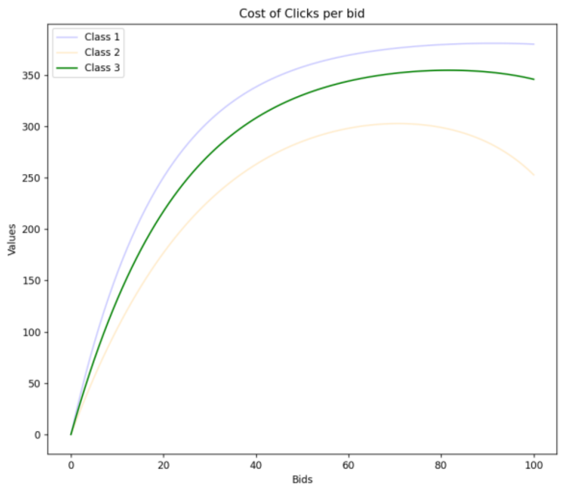
- **C3**: This class corresponds to old people that live in Italy: they are willingness to pay more to have the best accommodation and activities. Since Italy is a foreign country, the travel pack is considered as a luxury experience, so they perceive the high price as a good quality and experience. For this reason, the conversion probability is bigger when the price is increases).

  They aren't familiar with internet commerce, so the daily clicks is lower than the young people. The cost of the clicks is a little higher than C2 due to foreign ad services are more expensive.

  - **F1**: old people
  - **F2**: Foreign residence
  - **Conversion probabilities**: [0.25, 0.22, 0.24, 0.3, 0.35]
  - **Daily clicks per bids curve**: 50 * (1.0 - np.exp(-4 * x + 2 * x ** 3))
  - **Cost of click per bids curve**: return 400 * (1.0 - np.exp(-4 * x + 2 * x ** 3))







The reward function for our problem is:

$$Reward = \sum_{i=C1}^{C3} NC(bids) \cdot CP(price) \cdot (price - cost) \cdot CC(bids)$$

$$CP \equiv Conversion\ Probability$$
$$NC \equiv Number\ of\ daily\ Clicks$$
$$CC \equiv Cost\ of\ click$$

# STEP 1: LEARNING FOR PRICING

*Consider the case in which all the users belong to class C1. Assume that the curves related to the advertising part of the problem are known, while the curve related to the pricing problem is not. Apply the UCB1 and TS algorithms, reporting the plots of the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward.*

## Initial information

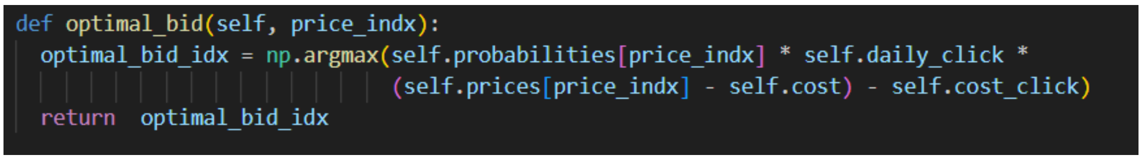
- We are only going to use class C1.
- The curves related to the advertising part are known.
- The curve related to the pricing problem is not.

With this information, the goal of the first step is to apply UCB1 and TS algorithm to estimate the conversion probabilities and find the best price.

Also, plot all the results in order to show: cumulative regret, cumulative reward, instantaneous regret, instantaneous reward.

## Environment

Due to the advertising part of the problem are known, we are going to select always the bid that maximize the reward. We are going to do this by implementing a function that finds this optimal value of bid:



```python
def optimal_bid(self, price_indx):
    optimal_bid_idx = np.argmax(self.probabilities[price_indx] * self.daily_click *
                                (self.prices[price_indx] - self.cost) - self.cost_click)
    return  optimal_bid_idx
```

$$Reward = NC(bids) \cdot CP(price) \cdot (price - cost) \cdot CC(bids)$$

This method simulates a round of interaction with the environment, given the `pulled_arm` (the chosen bid) by the agent. It calculates the reward, conversion rate, and number of clicks for the selected bid, for every round.

## TS_Learner

The way TS_learner works, is:

1. For every arm, we draw a sample according to the corresponding Beta distribution.
2. We choose the arm with the best sample. At every time t, we play the arm that:

$$a_t \leftarrow \arg max_{a \in A} \{\tilde{\theta}_a\}$$

```
def pull_arm(self):
  idx = np.argmax(np.random.beta(self.beta_parameters[:, 0], self.beta_parameters[:, 1])*(self.prices-self.cost*np.ones(self.n_arms)))
  return idx
```

3. We update the Beta distribution of the chosen arm according the observed realization, considering that in a given round the realization of the beta distribution are usually higher than and depends on the number of daily click. The beta distribution is updated considering the sum of the realizations :

$$(\alpha_{a_t,}\beta_{a_t}) \leftarrow (\alpha_{a_t,}\beta_{a_t}) + (x_{a_t t}, 1 - x_{a_t t})$$

```
def update(self, pulled_arm, conversion_rate, click_day_obs):
  self.t += 1
  self.update_observations(pulled_arm, conversion_rate)
  self.beta_parameters[pulled_arm, 0] = self.beta_parameters[pulled_arm, 0] + conversion_rate*click_day_obs
  self.beta_parameters[pulled_arm, 1] = self.beta_parameters[pulled_arm, 1] + 1*click_day_obs - conversion_rate*click_day_obs
```

The expected reward is calculated as the product of the sampled value from the Beta distribution and the difference between bid prices and costs.

## UCB1_Learner

The way UCB1_Learner works is:

1. In our case, we initially apply a function (find_first_zero) in order to select every arm before applying the algorithm, so every one is considered:

```
def find_first_zero(array):
    for i, num in enumerate(array):
        if num == 0:
            return i
    return -1  # Return -1 if zero is not found in the array
```

2. Every arm is associated with an upper confidence bound.

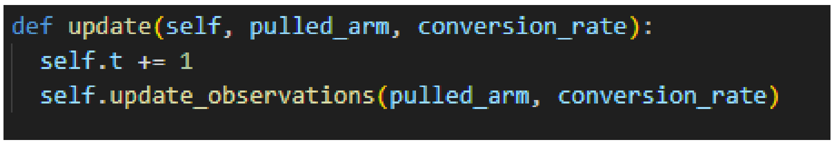
```
ucb_values = (self.collected_rewards_arm / self.arm_selections + np.sqrt(2*np.log(t) / self.arm_selections))*(self.prices-self.cost*np.ones(self.n_arms
```

3. At every round, the arm with the highest upper confidence bound is chosen.

```
idx = np.argmax(ucb_values)  # Select arm with highest UCB value
```

$$a_t \leftarrow \arg max_{a \in A} \left\{ \sqrt{\frac{2\log(t)}{n_a(t-1)}} \right\}$$

4. After having observed the realization of the reward of the arm, the upper confidence bound is uplated.



```
def update(self, pulled_arm, conversion_rate):
    self.t += 1
    self.update_observations(pulled_arm, conversion_rate)
```

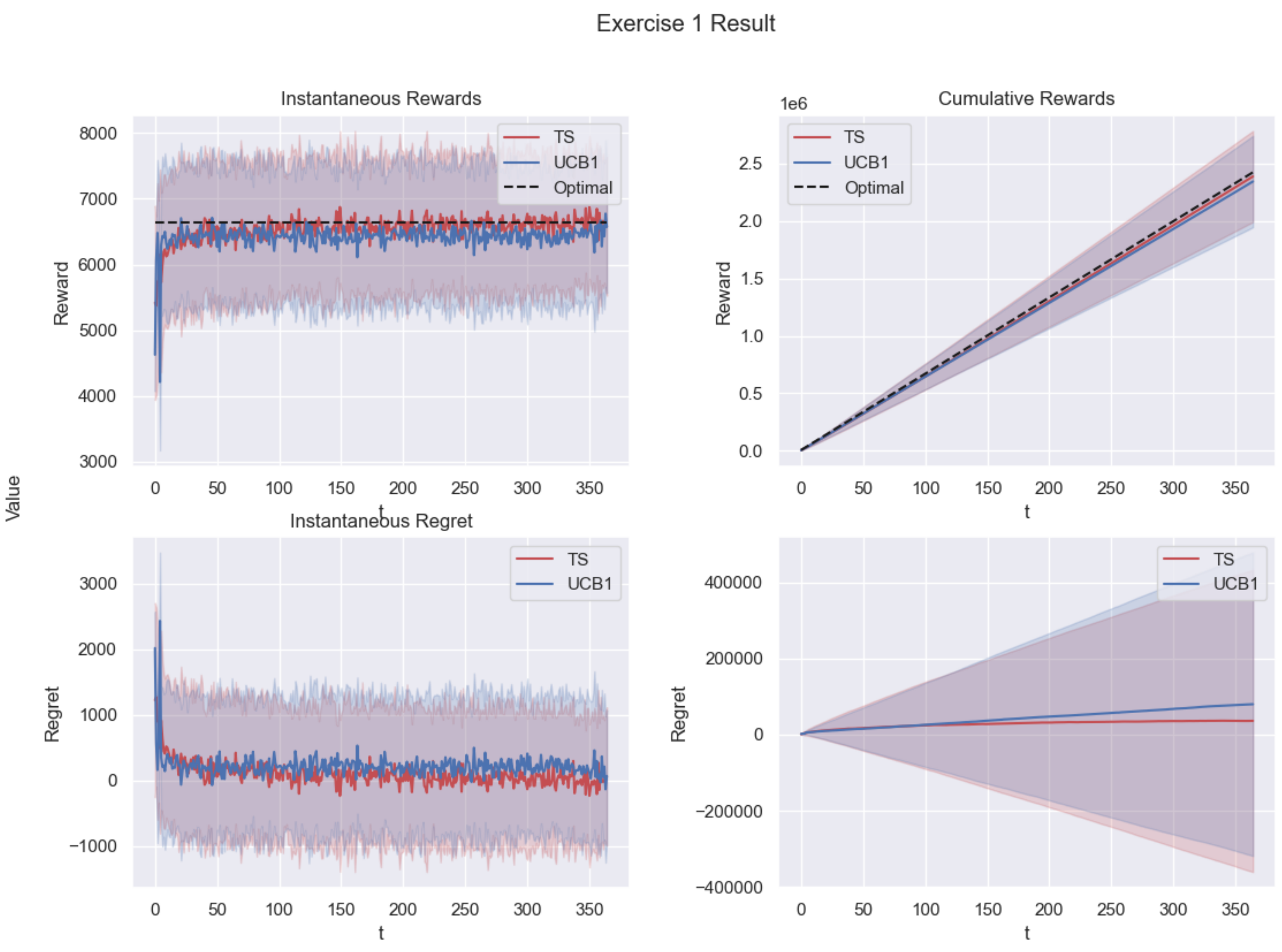
The pull_arm method returns the index of the selected arm.

## Results

The simulations was runed with the following parameters:

- n_experiments =200
- T = 365

And the results obtained were:



- As we can see in the Instantaneous Reward and Instantaneous regret graph, the TS algorithm achieves optimal solution.
- In the other hand, the UCB1 has a lower perform, but also achieves a great solution.
- These results were expected, since UCB1 algorithm works with upper confidence bounds, which leads to a lowest accuracy to find the best arm, since it’s more sensible to the number of samples. Using TS algorithm, we obtain an accurate result, thanks to a small variance of the Beta distribution.
- We can attribute the similar results in UCB1 and TS due to the predominance of one hand against the others.
- The two algorithms converge, but the TS goes faster.

# STEP 2: LEARNING FOR ADVERTISING

*Consider the case in which all the users belong to class C1. Assume that the curve related to the pricing problem is known while the curves related to the advertising problems are not. Apply the GP-UCB and GP-TS algorithms when using GPs to model the two advertising curves, reporting the plots of the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward.*

## Initial information

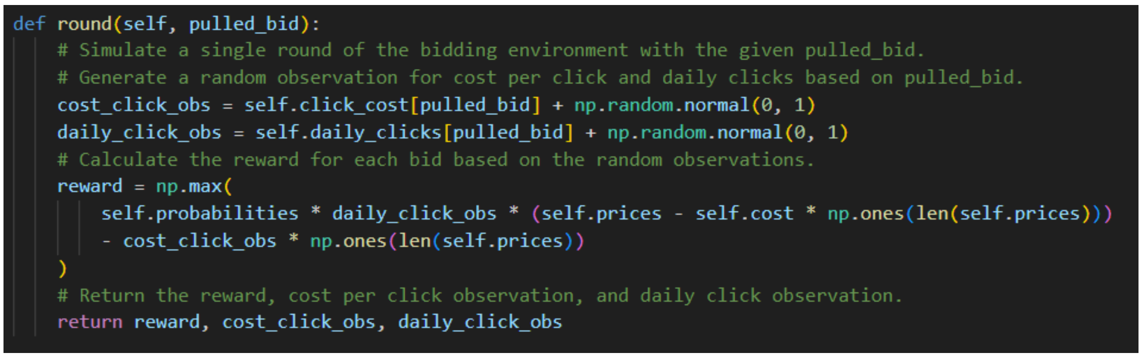
- We are only going to use class C1.
- The curves related to the advertising part are NOT known.
- The curve related to the pricing problem known.

With this information, the goal of the first step is to apply GP-UCB and GP-TS to find the advertising curves.

Also, plot all the results in order to show: cumulative regret, cumulative reward, instantaneous regret, instantaneous reward.

## Bidding Environment

In this case, we are working with a fixed price, so we are going to select the best price in terms of reward:



```python
def round(self, pulled_bid):
    # Simulate a single round of the bidding environment with the given pulled_bid.
    # Generate a random observation for cost per click and daily clicks based on pulled_bid.
    cost_click_obs = self.click_cost[pulled_bid] + np.random.normal(0, 1)
    daily_click_obs = self.daily_clicks[pulled_bid] + np.random.normal(0, 1)
    # Calculate the reward for each bid based on the random observations.
    reward = np.max(
        self.probabilities * daily_click_obs * (self.prices - self.cost * np.ones(len(self.prices)))
        - cost_click_obs * np.ones(len(self.prices))
    )
    # Return the reward, cost per click observation, and daily click observation.
    return reward, cost_click_obs, daily_click_obs
```

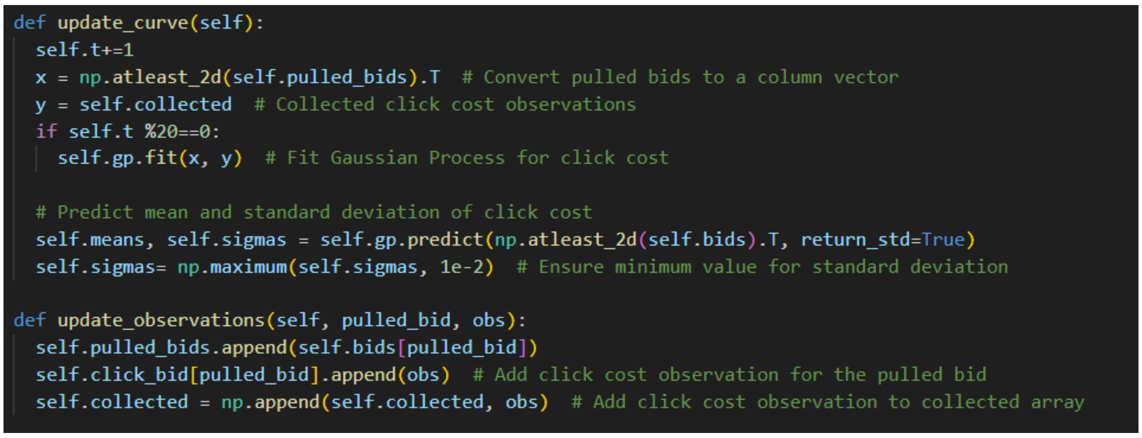
## Advertising curve

To calculate the advertising curves, we use Gaussian processes:

- We use them to store the information we obtain about number of clicks and cost of the clicks.
- Arms have correlation among them (smooth curves).

- We predict mean and standard deviation of click cost. The standard deviation is used to calculate the uncertainty.

we want to find the best bid to maximize the total reward.



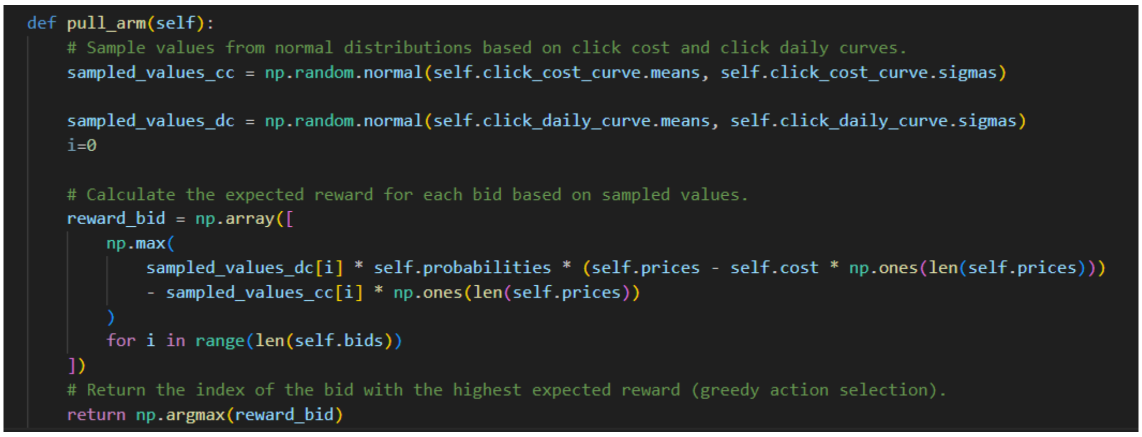
```python
def update_curve(self):
    self.t+=1
    x = np.atleast_2d(self.pulled_bids).T  # Convert pulled bids to a column vector
    y = self.collected  # Collected click cost observations
    if self.t %20==0:
        self.gp.fit(x, y)  # Fit Gaussian Process for click cost

    # Predict mean and standard deviation of click cost
    self.means, self.sigmas = self.gp.predict(np.atleast_2d(self.bids).T, return_std=True)
    self.sigmas= np.maximum(self.sigmas, 1e-2)  # Ensure minimum value for standard deviation

def update_observations(self, pulled_bid, obs):
    self.pulled_bids.append(self.bids[pulled_bid])
    self.click_bid[pulled_bid].append(obs)  # Add click cost observation for the pulled bid
    self.collected = np.append(self.collected, obs)  # Add click cost observation to collected array
```

After that, the method computes the reward for each price level by using the reward formula:

$$Reward = NC(bids) \cdot CP(price) \cdot (price - cost) \cdot CC(bids)$$

## GPTS Learner advert

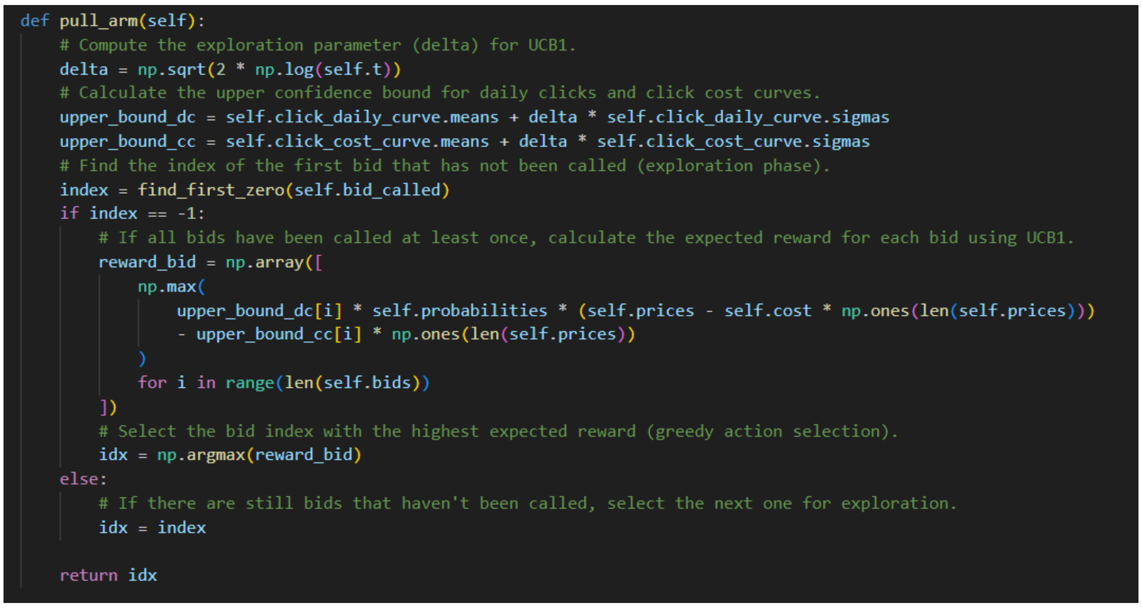
The TS algorithm for this case, as said before, works calculating the Gaussian distribution instead of the Beta distribution. The rest of the algorithm remains the same:



```python
def pull_arm(self):
    # Sample values from normal distributions based on click cost and click daily curves.
    sampled_values_cc = np.random.normal(self.click_cost_curve.means, self.click_cost_curve.sigmas)

    sampled_values_dc = np.random.normal(self.click_daily_curve.means, self.click_daily_curve.sigmas)
    i=0

    # Calculate the expected reward for each bid based on sampled values.
    reward_bid = np.array([
        np.max(
            sampled_values_dc[i] * self.probabilities * (self.prices - self.cost * np.ones(len(self.prices)))
            - sampled_values_cc[i] * np.ones(len(self.prices))
        )
        for i in range(len(self.bids))
    ])
    # Return the index of the bid with the highest expected reward (greedy action selection).
    return np.argmax(reward_bid)
```

## GPUCB1 Learner advert

Works as the previous UCB1 algorithm, but the upper bounds are calculated also with Gaussian distributions:



```python
def pull_arm(self):
    # Compute the exploration parameter (delta) for UCB1.
    delta = np.sqrt(2 * np.log(self.t))
    # Calculate the upper confidence bound for daily clicks and click cost curves.
    upper_bound_dc = self.click_daily_curve.means + delta * self.click_daily_curve.sigmas
    upper_bound_cc = self.click_cost_curve.means + delta * self.click_cost_curve.sigmas
    # Find the index of the first bid that has not been called (exploration phase).
    index = find_first_zero(self.bid_called)
    if index == -1:
        # If all bids have been called at least once, calculate the expected reward for each bid using UCB1.
        reward_bid = np.array([
            np.max(
                upper_bound_dc[i] * self.probabilities * (self.prices - self.cost * np.ones(len(self.prices)))
                - upper_bound_cc[i] * np.ones(len(self.prices))
            )
            for i in range(len(self.bids))
        ])
        # Select the bid index with the highest expected reward (greedy action selection).
        idx = np.argmax(reward_bid)
    else:
        # If there are still bids that haven't been called, select the next one for exploration.
        idx = index

    return idx
```

The outcome of both algorithms are expected reward based on the sampled click cost and daily clicks. This expected reward is compare with each bid and selected the maximum value.

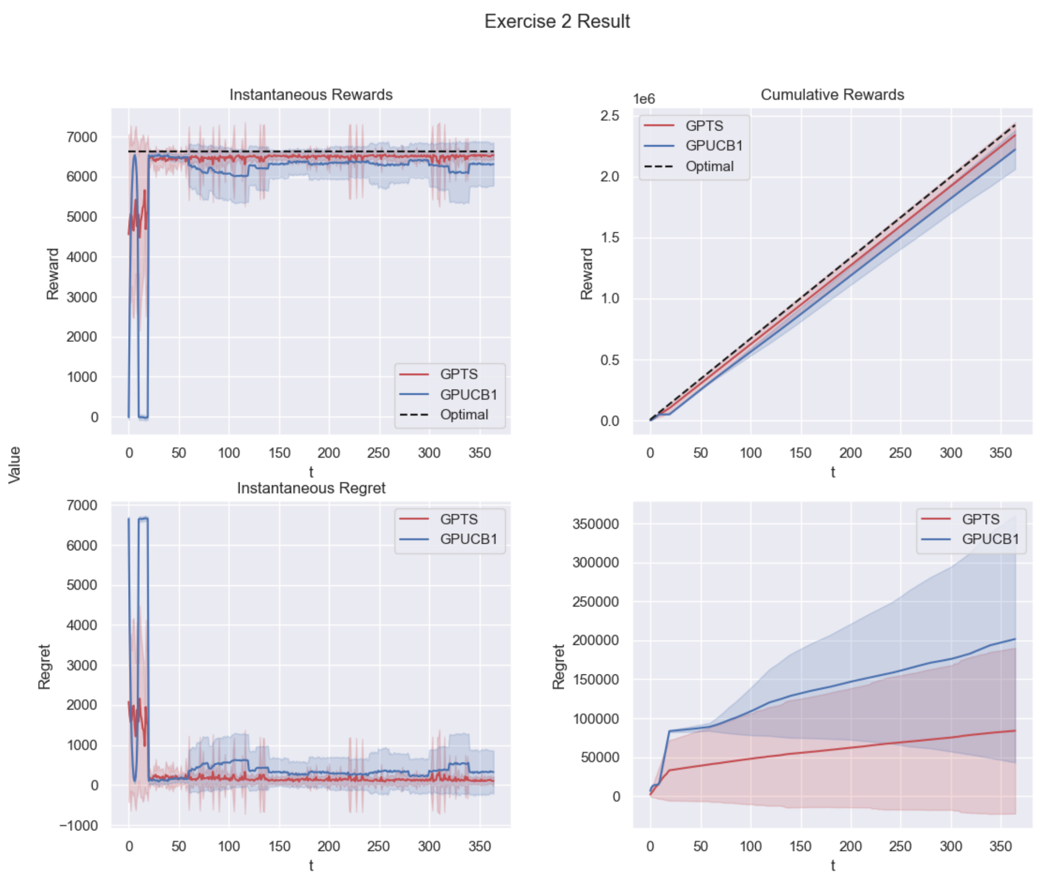
$$Reward = NC(bids) \cdot CP(price) \cdot (price - cost) \cdot CC(bids)$$

# RESULTS

The simulations was ran with the following parameters:

- n_experiments =50
- T = 365

And the results obtained were:



- As we can see in the Instantaneous Reward and Instantaneous regret graph, the TS algorithm achieves optimal solution.
- In the other hand, the UCB1 has a lower perform. In some round intervals the values decrease, making a “step” form.
- At the beginning, the GPUCB1 algorithm gives has a very bad reward values, since the upper confidence is more sensible at the beginning of the experiment.
- The two algorithms converge, but the TS goes faster.
- Both curves present a hedgy form, this is due on the fact that for computational purpose we are recalibrating the two advertising curves only once each 20 rounds.

## STEP 3: LEARNING FOR JOINT PRICING AND ADVERTISING

*Consider the case in which all the users belong to class C1, and no information about the advertising and pricing curves is known beforehand. Apply the GP-UCB and GP-TS algorithms when using GPs to model the two advertising curves, reporting the plots of the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward.*

## Initial information

- We are only going to use class C1.
- The curves related to the advertising and pricing part are NOT known.

With this information, the goal of the first step is to apply the previous algorithms to find the advertising curves.

Also, plot all the results in order to show: cumulative regret, cumulative reward, instantaneous regret, instantaneous reward.

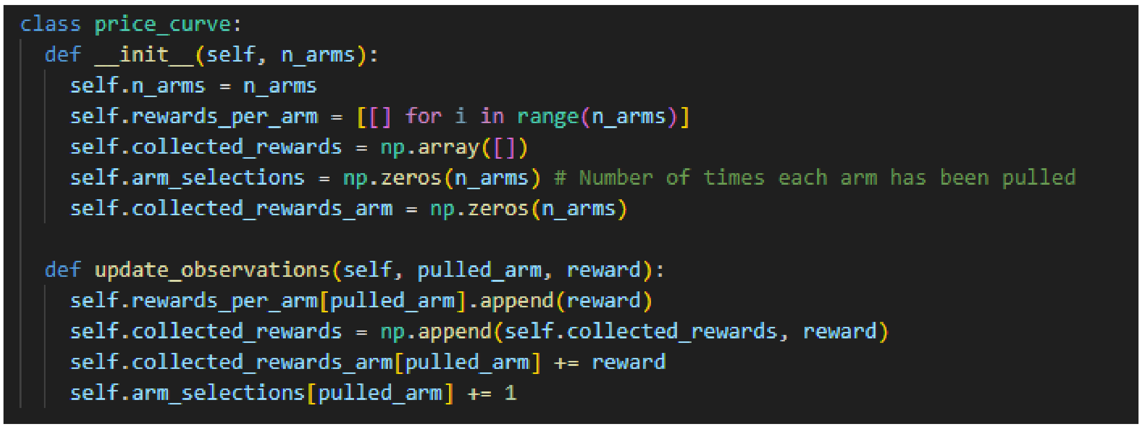
## Procedure

As the project proposal suggest us, we are going to apply the following procedure:

1. First, we are going to optimize the pricing problem to find the best price.
2. Given this price, we optimize the advertising problem to find the suggested bid.

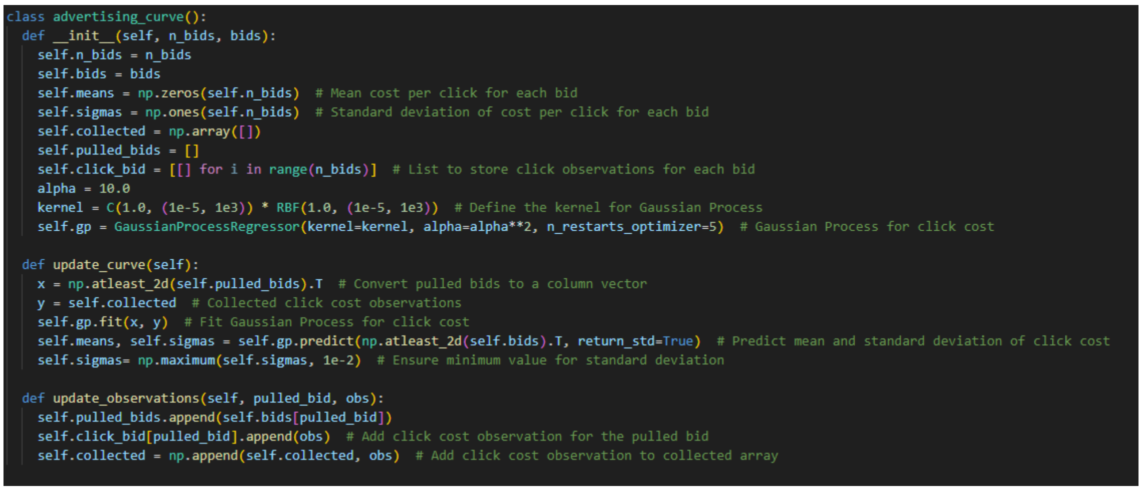
The procedure of the single pricing and advertising problems are the same as the previous steps.

## Pricing problem



```python
class price_curve:
  def __init__(self, n_arms):
    self.n_arms = n_arms
    self.rewards_per_arm = [[] for i in range(n_arms)]
    self.collected_rewards = np.array([])
    self.arm_selections = np.zeros(n_arms) # Number of times each arm has been pulled
    self.collected_rewards_arm = np.zeros(n_arms)

  def update_observations(self, pulled_arm, reward):
    self.rewards_per_arm[pulled_arm].append(reward)
    self.collected_rewards = np.append(self.collected_rewards, reward)
    self.collected_rewards_arm[pulled_arm] += reward
    self.arm_selections[pulled_arm] += 1
```

## Advertising problem



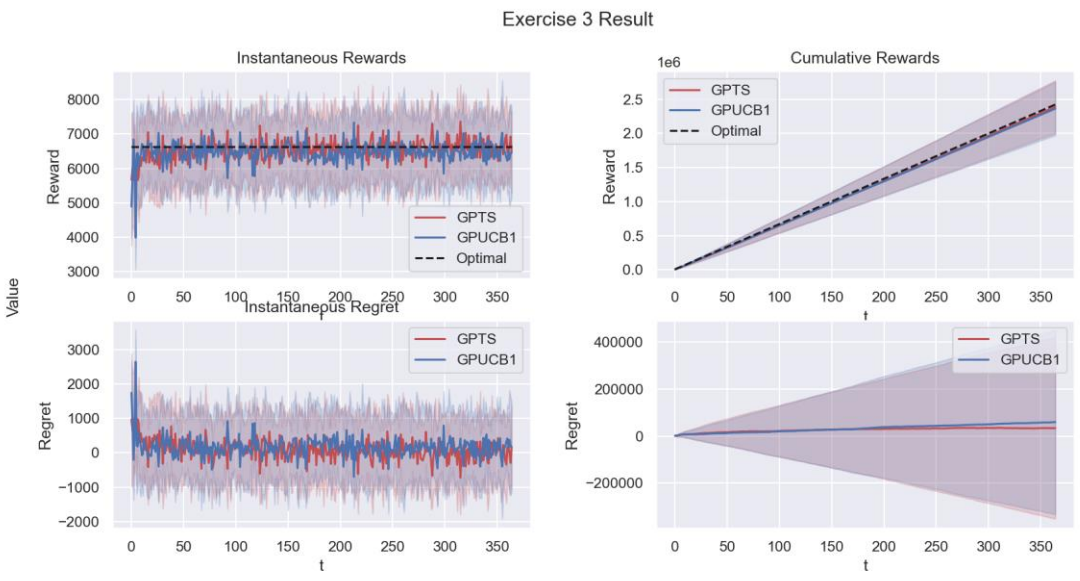
```python
class advertising_curve():
  def __init__(self, n_bids, bids):
    self.n_bids = n_bids
    self.bids = bids
    self.means = np.zeros(self.n_bids)  # Mean cost per click for each bid
    self.sigmas = np.ones(self.n_bids)  # Standard deviation of cost per click for each bid
    self.collected = np.array([])
    self.pulled_bids = []
    self.click_bid = [[] for i in range(n_bids)]  # List to store click observations for each bid
    alpha = 10.0
    kernel = C(1.0, (1e-5, 1e3)) * RBF(1.0, (1e-5, 1e3))  # Define the kernel for Gaussian Process
    self.gp = GaussianProcessRegressor(kernel=kernel, alpha=alpha**2, n_restarts_optimizer=5)  # Gaussian Process for click cost

  def update_curve(self):
    x = np.atleast_2d(self.pulled_bids).T  # Convert pulled bids to a column vector
    y = self.collected  # Collected click cost observations
    self.gp.fit(x, y)  # Fit Gaussian Process for click cost
    self.means, self.sigmas = self.gp.predict(np.atleast_2d(self.bids).T, return_std=True)  # Predict mean and standard deviation of click cost
    self.sigmas= np.maximum(self.sigmas, 1e-2)  # Ensure minimum value for standard deviation

  def update_observations(self, pulled_bid, obs):
    self.pulled_bids.append(self.bids[pulled_bid])
    self.click_bid[pulled_bid].append(obs)  # Add click cost observation for the pulled bid
    self.collected = np.append(self.collected, obs)  # Add click cost observation to collected array
```

## Results

The simulations were ran with the following parameters:

- n_experiments =50
- T = 365

And the results obtained were:



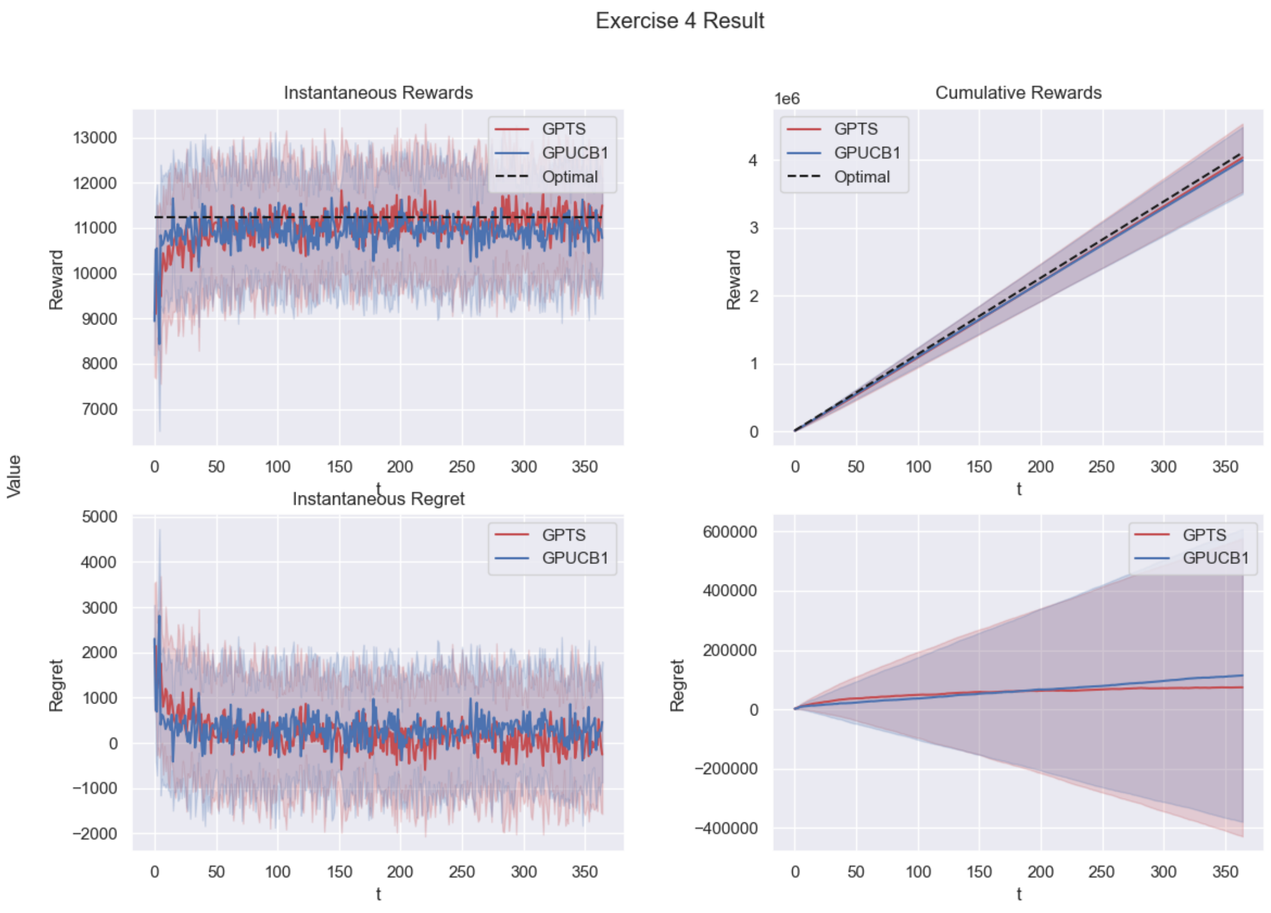
- As we can see in the Instantaneous Reward and Instantaneous regret graph, the TS algorithm achieves optimal solution.
- In the other hand, the UCB1 has a lower perform.

- Like the other steps, this result was expected. But in this case, now we can distinguish between the results of the two algorithms. This is because in this case the difference between the rewards of the arms is more competitive.
- The two algorithms converge, but the TS goes faster and with a better result.

## STEP 4: CONTEXTS AND THEIR GENERATION

*Consider the case in which there are three classes of users (C1, C2, and C3), and no information about the advertising and pricing curves is known beforehand. Consider two scenarios. In the first one, the structure of the contexts is known beforehand. Apply the GP-UCB and GP-TS algorithms when using GPs to model the two advertising curves, reporting the plots with the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward. In the second scenario, the structure of the contexts is not known beforehand and needs to be learnt from data. Important remark: the learner does not know how many contexts there are, while it can only observe the features and data associated with the features. Apply the GP-UCB and GP-TS algorithms when using GPs to model the two advertising curves paired with a context generation algorithm, reporting the plots with the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward. Apply the context generation algorithms every two weeks of the simulation. Compare the performance of the two algorithms --- the one used in the first scenario with the one used in the second scenario. Furthermore, in the second scenario, run the GP-UCB and GP-TS algorithms without context generation, and therefore forcing the context to be only one for the entire time horizon, and compare their performance with the performance of the previous algorithms used for the second scenario.*

## RESULTS



For lack of timing, we were able to run only the first scenario in which the contexts are known beforehand. Please find in the image above the results. The reward is defined in this scenario as the sum of the different reward for each class.

## STEP 5: DEALING WITH NON-STATIONARY ENVIRONMENTS WITH TWO ABRUPT CHANGES

*Consider the case in which there is a single-user class C1. Assume that the curve related to the pricing problem is unknown while the curves related to the advertising problems are known. Furthermore, consider the situation in which the curves related to pricing are non-stationary, being subject to seasonal phases (3 different phases spread over the time horizon). Provide motivation for the phases. Apply the UCB1 algorithm and two non-stationary flavors of the UCB1 algorithm defined as follows. The first one is passive and exploits a sliding window, while the* *second one is active and exploits a change detection test. Provide a sensitivity analysis of the parameters employed in the algorithms, evaluating different values of the length of the sliding window in the first case and different values for the parameters of the change detection test in the second case. Report the plots with the average (over a sufficiently large number of runs) value and standard deviation of the cumulative regret, cumulative reward, instantaneous regret, and instantaneous reward. Compare the results of the three algorithms used.*

## Initial information

- We are only going to use class C1.
- The curves related to the advertising problem are known.
- The curves related to the pricing problem are unknown, and they are **NON-STARIONARY** with three different phases.

With this information, the goal of the first step is to apply UCB1 algorithm to estimate the curve of the prizing model.

## UCB1 Algorithm

The UCB1 algorithm is the same as the one used in step1, for this reason we skip the description of the algorithm in this section. Please refer to the section 1 for more information.

## SW UCB1 Algorithm

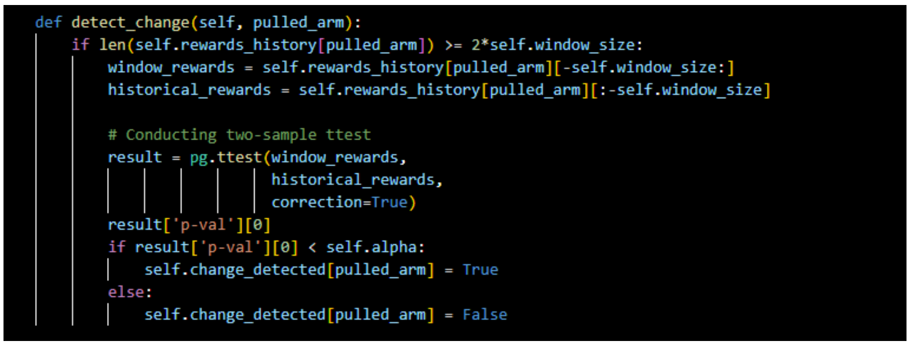
The "Sliding-Window UCB1" (SW UCB1) method is a version of the original UCB1 algorithm designed to handle non-stationary situations with changing rewards of arms. It is widely employed in multi-armed bandit issues where the reward distributions of the arms may vary and the algorithm must adapt to these changes. The main idea underlying SW UCB1 is to maintain track of recent awards using a sliding time frame and update the UCB estimations depending on the rewards inside that window. This sliding window method enables the algorithm to focus on recent data while discarding earlier, perhaps obsolete data. Utilizing a sliding window of recent data enables it to avoid reliance on outdated information, resulting in improved performance in non-stationary environments. Nonetheless, choosing the optimal window size W is crucial as it strikes a balance between adaptability and the sufficiency of data required for dependable estimates. For this reason in the further section we will present a sensitivity test on the parameter W.

## CD UCB1 Algorithm

The change detector UCB1 algorithm is a version of the simplest UCB1 algorithm, where a change detector test is applied to verify if the means of the rewards are significantly different in the recent output compared to the historical one.

In our case, we are interested in identifying changes in the underlying reward distribution of the bandit arms while utilizing the t-Student test as a change detector in the UCB1 algorithm. The t-Student test is a statistical hypothesis test used to compare the means of two samples and determine if they vary substantially. We will implement this test on a the most recent portion of the reward vs the historical one.

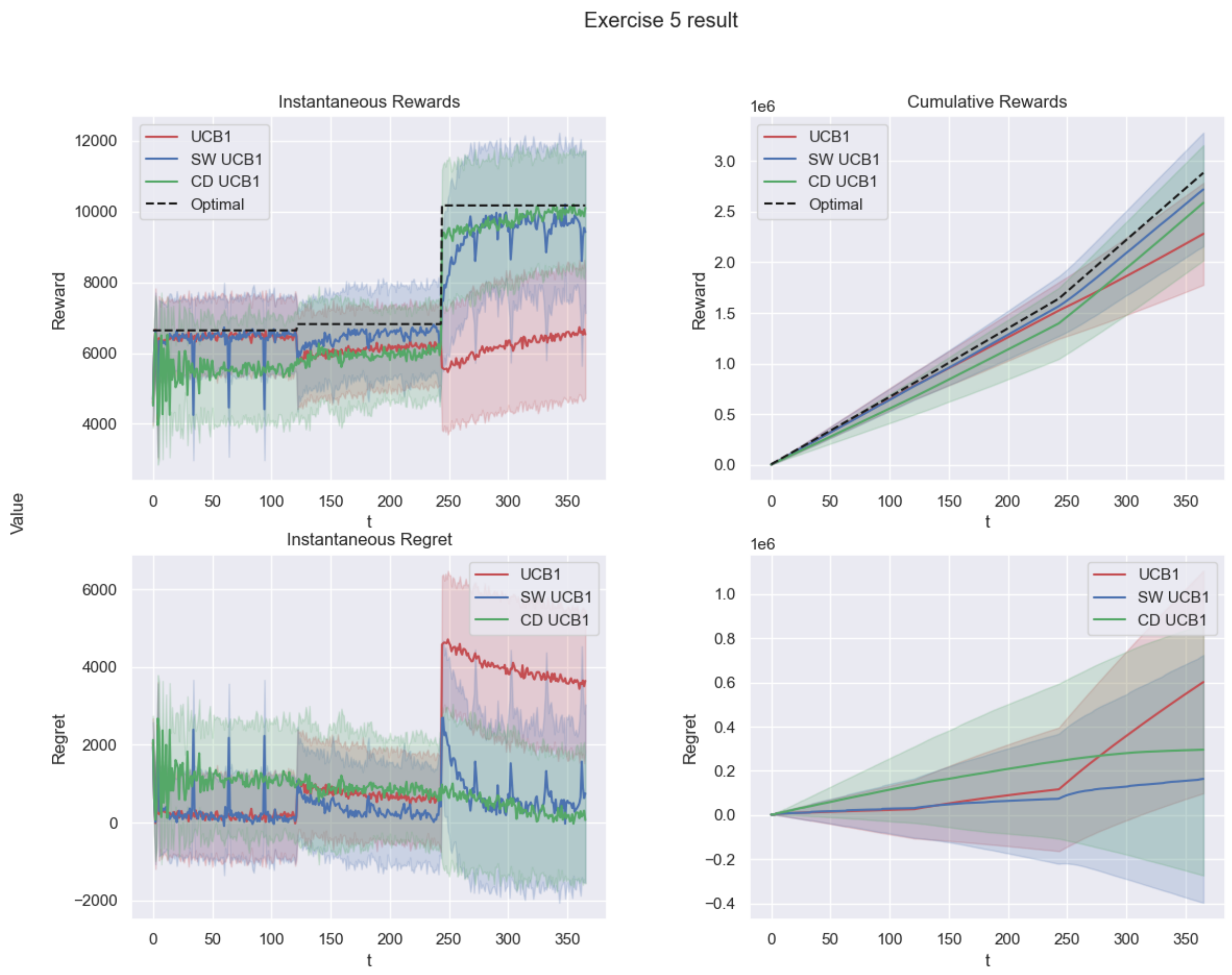
For this reason, the CD UCB1 algorithm has 2 inputs, the length of the window W and the parameter alpha, i.e., the parameter of the t student test. Please find below the part of the code in which the test is applied.



```
def detect_change(self, pulled_arm):
    if len(self.rewards_history[pulled_arm]) >= 2*self.window_size:
        window_rewards = self.rewards_history[pulled_arm][-self.window_size:]
        historical_rewards = self.rewards_history[pulled_arm][:-self.window_size]

        # Conducting two-sample ttest
        result = pg.ttest(window_rewards,
                          historical_rewards,
                          correction=True)
        result['p-val'][0]
        if result['p-val'][0] < self.alpha:
            self.change_detected[pulled_arm] = True
        else:
            self.change_detected[pulled_arm] = False
```

## Results

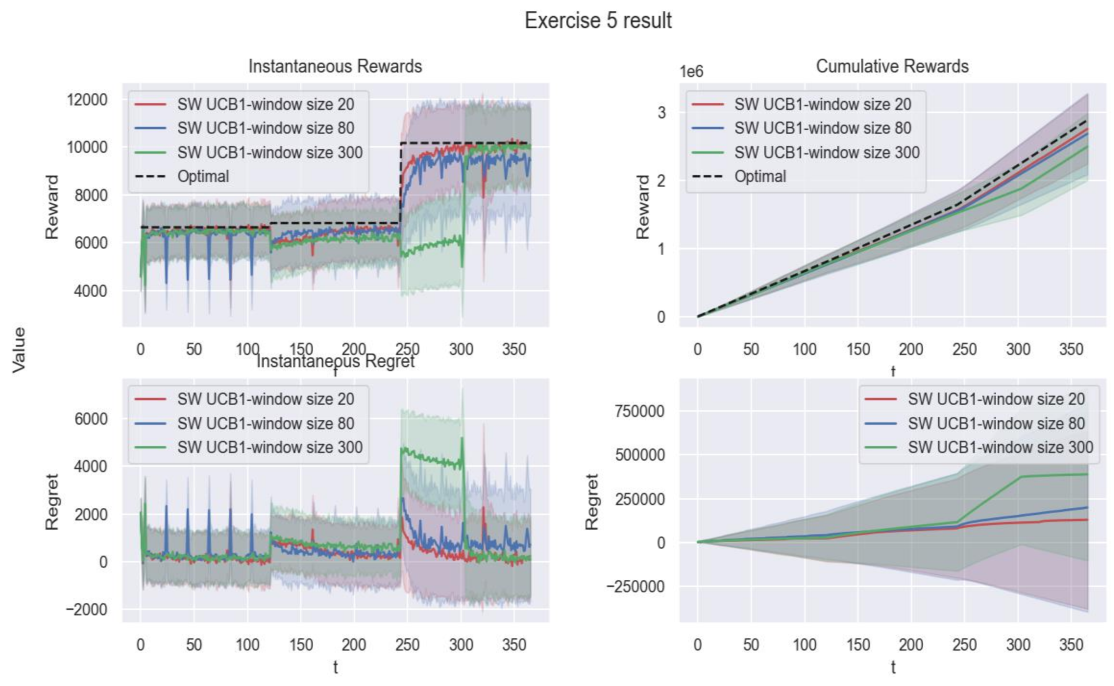
As shown in the image below given the non-stationarity nature of the problem, the UCB1 method failed to correctly capture the change in the optimal reward curve. While the other two method perform very good, being able to immediately spot the abrupt changes and change accordingly their distribution.



## Sensitivity test

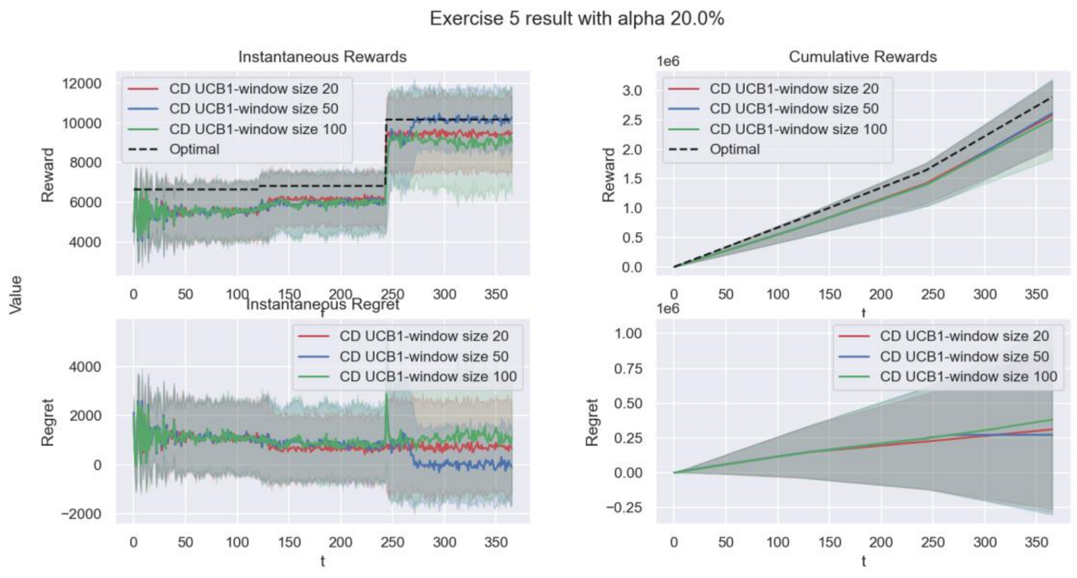
SW UCB1 considers W as the size of the sliding windows, whereas CD UCB1 contains two parameters, the test parameter alpha and the window size W.

We tried the SW UCB1 algorithm with three distinct W parameters: 20, 80, and 300. The approach with W=20 performs better than the one with W=300, as predicted, because the algorithm is too slow to detect changes in the ideal reward curve.



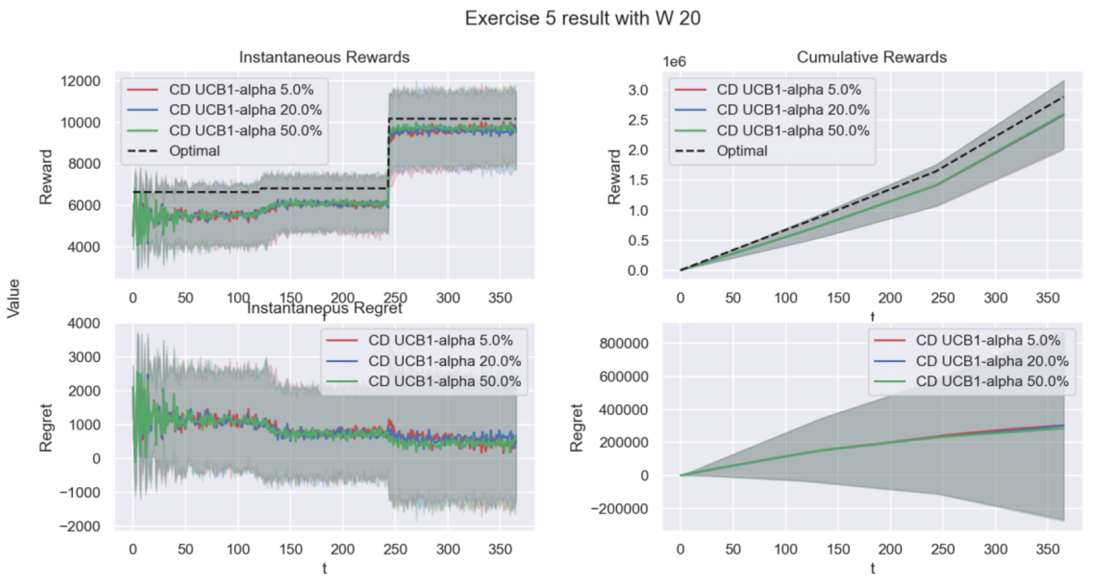
In the same way, we simulated the CD UCB1 method with three alternative options for W, 20, 50, and 100, while leaving the alpha parameter fixed at 20%. Even in this scenario, the algorithm with W=20 and 50 performs best, whereas the algorithm with W=100 performs worst since the system is too slow to detect changes in the ideal reward curve.

Please see the image below for the outcome.



Finally, we repeated the technique with the value W set to 20 and the parameters alpha set to 0,05, 0,2, and 0.5. The greater the alpha value, the more likely it is that a change is identified even if there is no change in the mean of the reward.

As shown in the graph below, changing the parameter alpha has little effect on the outcome.



# STEP 6: DEALING WITH NON STATIONARY ENVIRONMENTS WITH MANY ABRUPT CHANGES

*Develop the EXP3 algorithm, which is devoted to dealing with adversarial settings. This algorithm can be also used to deal with non-stationary settings when no information about the specific form of non-stationarity is known beforehand. Consider a simplified version of Step 5 in which the bid is fixed. First, apply the EXP3 algorithm to this setting. The expected result is that EXP3 performs worse than the two non-stationary versions of UCB1. Subsequently, consider a different non-stationary setting with a higher non-stationarity degree. Such a degree can be modeled by having a large number of phases that frequently change. In particular, consider 5 phases, each one associated with a different optimal price, and these phases cyclically change with a high frequency. In this new setting, apply EXP3, UCB1, and the two non-stationary flavors of UBC1. The expected result is that EXP3 outperforms the non-stationary version of UCB1 in this setting.*

## Initial information

EXP3 is an algorithm tailored for the adversarial bandit setting. In each round, it chooses an arm based on a random draw from a probability distribution calculated in the previous step. EXP3's adversarial nature introduces learning tendencies. When an arm yields a high reward, its weight increases, leading to a higher probability of being selected in subsequent rounds.

The probability of selecting arm i at round t is governed by the formula:

$$pi(t) = (1 - \gamma) * \frac{w_i(t)}{\Sigma w_j\,(t)} + \frac{\gamma}{K}.$$

Here, K represents the number of arms, and w_i(t) denotes the weight associated with arm i at round t. The hyperparameter γ, ranging between 0 and 1, influences the probability distribution.

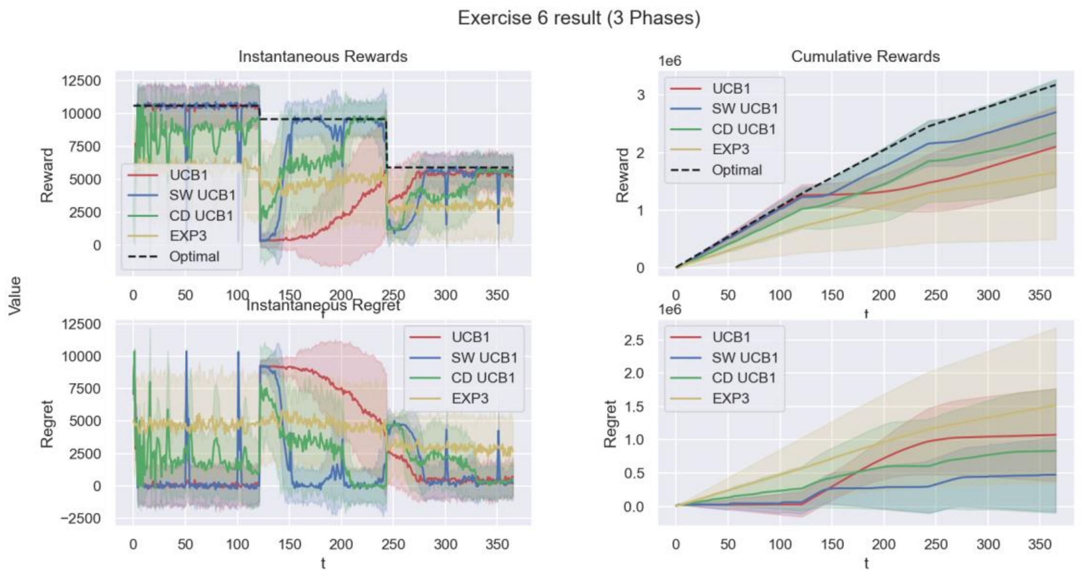
To update the weight of arm i at round t, the algorithm employs the formula:

$$w_i\,(t+1) = w_i(t) * e^{\gamma/K * \frac{x_{it}}{p_{it}}},$$

where xit / pit represents the expected reward obtained from arm i at time t. It is essential to initialize the weights to 1 for each arm, and the rewards should be rescaled to fall within the interval [0, 1]. The value of γ dictates the balance between a more uniform probability distribution (closer to 1) and a distribution based on rewards obtained (closer to 0).

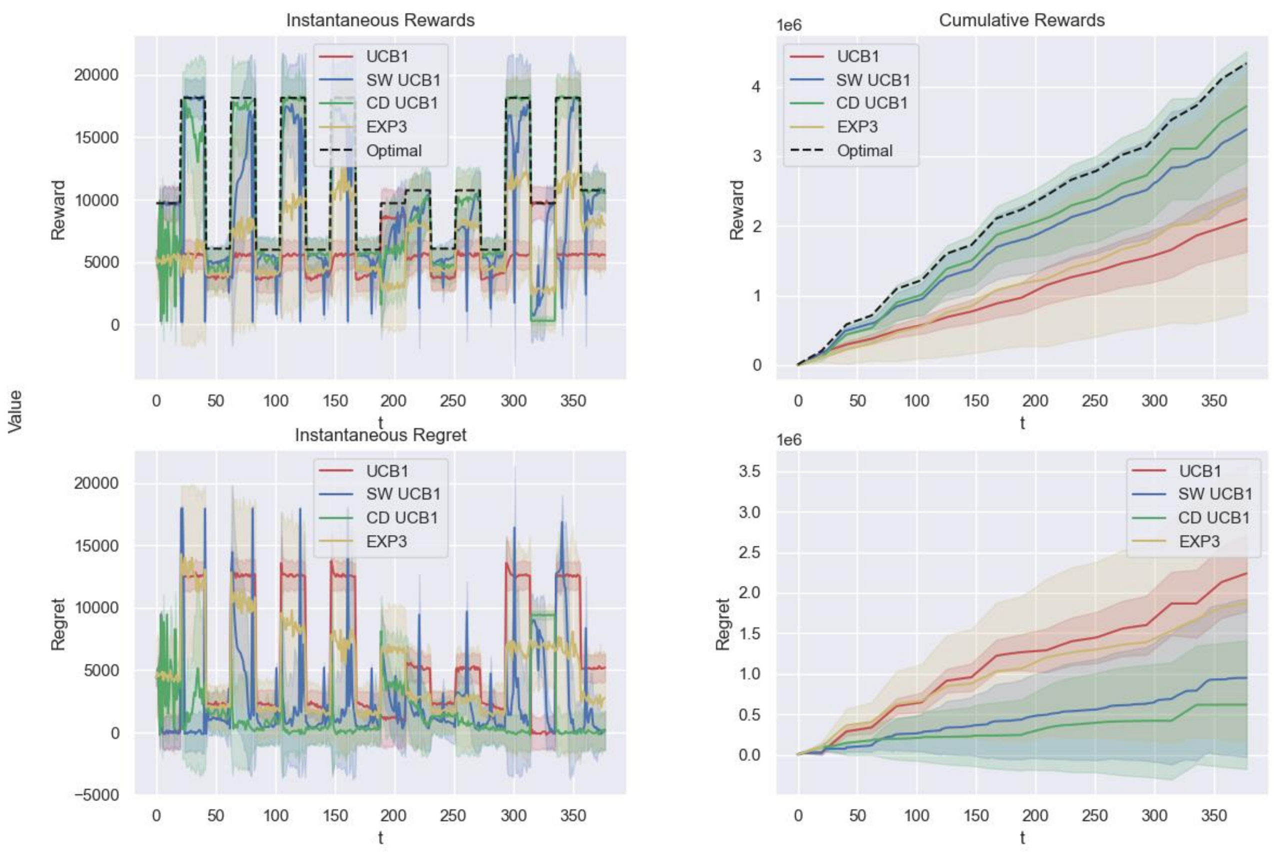
## Results

We examine EXP3's performance under the Step 5 scenario, where there are two transitions, each lasting roughly 120 rounds, resulting in three phases. The EXP3 method, as expected, does not perform well in a context with only two modifications over 365 cycles.



If we consider instead in an high frequency scenario the model performs quite well, in our scenario we are considering a high number of changes in the pricing curve with the same length curve of 365 rounds. Unfortunately, we were expecting that algorithm was able to outperform the two precedent algorithms in an high frequency scenario, but as showed in the image below this was not the case in our implementation. For a lack of time we were not able to solve the problem.

Exercise 6 result (High Frequency)



POLITECNICO
MILANO 1863