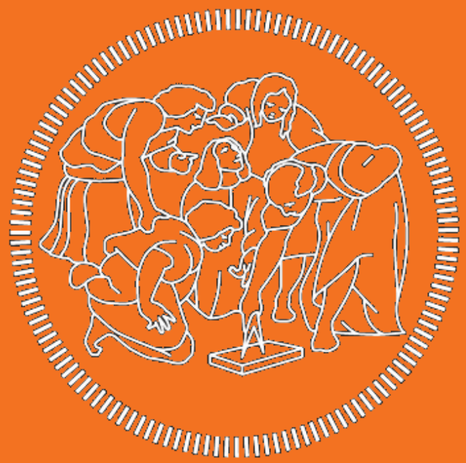
Online Learning Applications

“Pricing and Advertising strategy for e-Commerce of Flight Booking”

A.A. 2022-2023

Filippo Lazzarin, Alberto Pirillo,
Michele Simeone, Simone Tognocchi



POLITECNICO
MILANO 1863

Team

Michele Simeone

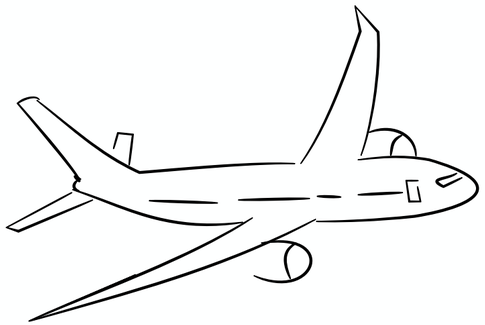
Simone Tognocchi

Alberto Pirillo

Filippo Lazzarin

Overview

- Problem setting
- Environment design
- Clairvoyant algorithm



Steps

1. Learning for pricing
2. Learning for advertising
3. Learning for joint pricing and advertising
4. Contexts and their generation
5. Dealing with non-stationary environments with two abrupt changes
6. Dealing with non-stationary environments with many abrupt changes

Problem setting

An **airline company** which sells plane tickets in its **e-commerce** platform wants to maximize its profit by dynamically learning an optimal pricing and advertising strategy.

Pricing: the same ticket could be sold at 5 different prices. The company wants to know the best one, i.e. the one that maximizes the **product of the conversion rate by the price**. The conversion rate represents whether a user who has seen the ticket will buy it.

Standard **Multi-Armed Bandits (MABs)** algorithms can be used to learn the conversion rates in an online fashion efficiently.

Advertising: the company wants to develop an **online advertising campaign** so that many users will discover it and see the services it offers by visiting the webpage of a **publisher.** The company has to select a **bid** (the maximum amount it is willing to pay the publisher). Selecting a higher bid will result in more users seeing the advertisement but also more costs charged by the publisher to the company.

Gaussian Process (GP) Bandits algorithms can be used to dynamically learn the optimal bid while minimizing the loss in the meantime.

Environment Design Classes

Based on the information that we retrieve from the purchase of the ticket we can divide customers into different segments.

The features that we have chosen to classify the users are:

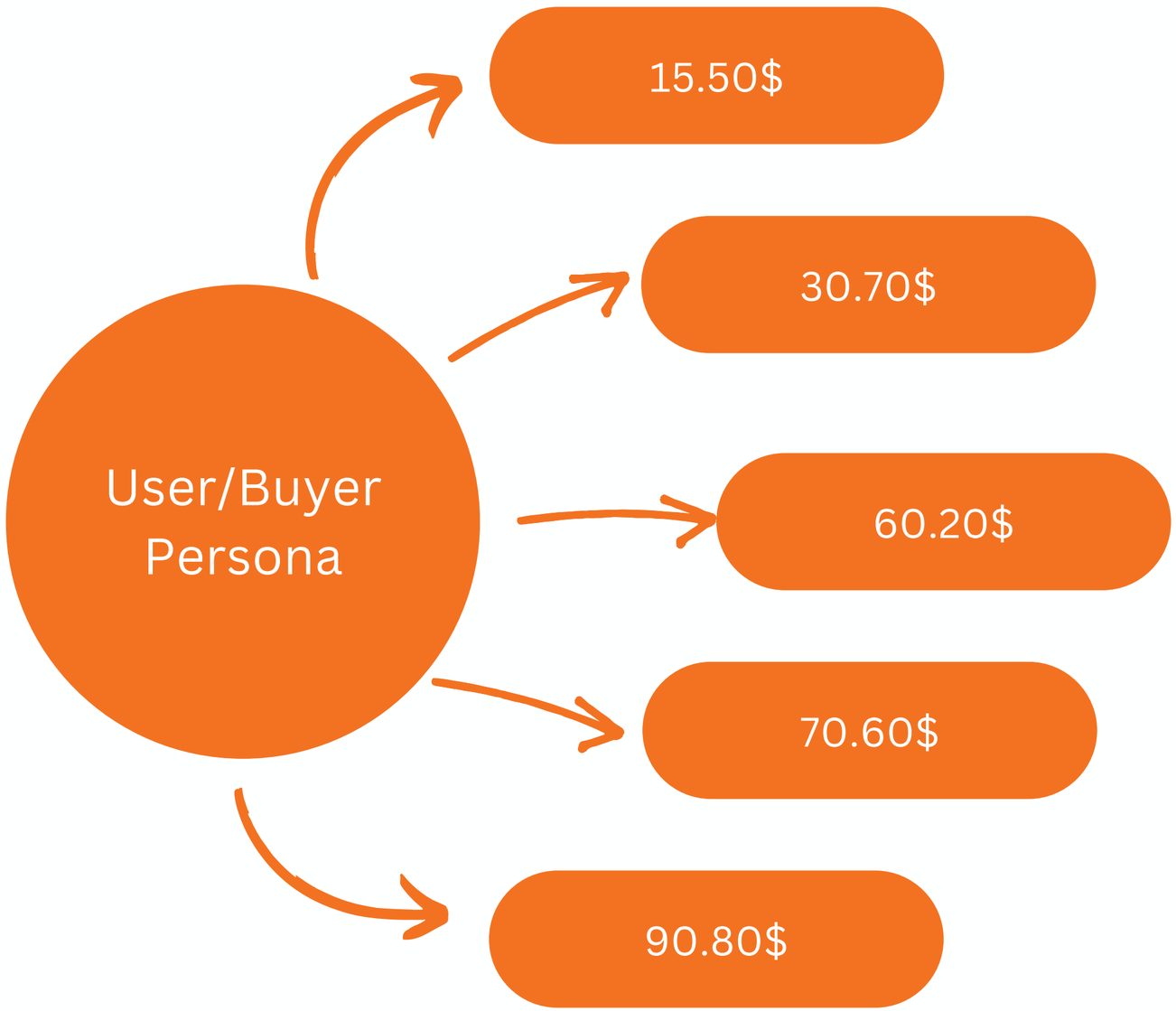
- Whether a customer chooses to buy **checked luggage**
- Whether a customer chooses to buy a **first-class ticket**

Based on these features we created three different contexts:

- First class of users is represented by those **under 25** years
- Second class by adult users with an age in the **range of 25-40** years
- Third class by those **over 40** years

The **under 25** do not pick up any feature for their ticket.
The **adult (25-40)** users pick up just one between checked luggage and a first-class ticket.
The **over 40** choose both features when purchasing the ticket.

Classes and Prices



Environment Design
Conversion Rates

The flight market is affected by **seasonality** which subsequently influences the conversion rates.

In general, we can split the time horizon (**365 days**) into **three periods**:

- The **winter period** goes from October to January
- The **spring period** from February to May
- The **summer period** from June to September

Usually, tickets tend to have the **highest conversion rates during summer**, while conversion rates tend to be slightly lower during winter and are the **lowest in the spring**.

When we decided on the conversion rates we tried to be as faithful as possible to **real-world** case scenarios while using values that allow for easily visualizable results.

Environment Design
Prices and Bids

We propose **5 different ticket prices** ordered in ascending order based on the inclusion of the checked luggage and the first-class ticket features. The possible ticket prices are the following: 15.50$, 30.70$, 60.20$, 70.60$, 90.80$.
Every user class has a **Bernoulli distribution** associated with each price.

Moreover, the three proposed classes **differ** in terms of:

- the function that expresses the number of **daily clicks** as the **bid varies**
- the function that assigns the **cumulative daily cost** of the clicks as the **bid varies**

Both the functions are **concave curves** obtained by using an exponential function with different constants and then by adding **Gaussian noise**.

An example of these functions can be seen in the following plots.

Environment Design
Prices and Bids

These are not the actual functions defined in the environment but rather the **noisy observations** that the learners see when they interact with the environment. **Gaussian noise** was added to the concave curves.

Clairvoyant algorithm

The **objective function** to maximize is defined as the **reward**.

- For one class the **reward** is defined as:

(number of daily clicks * conversion rate * margin) - cumulative daily cost

- The **margin** is the difference between the price at which the product is sold and its corresponding cost. For simplicity, we consider a cost of 0 for each product.
- For **multiple classes**, the reward will be the **sum** of the rewards provided by the single classes.

We have approximated the continuous set of bids with a finite set of 100 bids.

The optimization algorithm works as follows:

- Find the best price for every single class, independently from the others
- Optimize the bid for each class independently from the other classes

Therefore, the algorithm requires **two exhaustive searches:**

- Over the prices for every class
- Over the bids for every class

Thus, the algorithm runs in linear time in the number of prices, bids, and classes.

Experiment parallelization

When working with **stochastic processes**, it is required to run multiple simulations of the same scenario in order to **smooth out the stochasticity**. We refer to these simulations as *experiments*.

The **proper number of experiments** depends on the environments and on the algorithms used. We found that pricing algorithms (e.g. UCB1, TS) required much more experiments w.r.t. advertising algorithms (e.g. GPU-UCB, GP-TS).

Every experiment is **independent** from the others. Therefore, we can **run multiple experiments in parallel** on different CPU cores, using the *map* function of the *multiprocessing* Python module.

Essentially, we **encapsulated the entirety of an experiment inside of a single function**. Then, the function is executed in parallel on different cores. In the end, the results are collected in a common data structure.

Observed speedups are in the range of **5x to 7x** depending on the number of available CPU cores.
Parallelization was the reason why we were able to perform this many experiments.

JSON Environments

In order to duplicate the least amount of code possible and ensure that all experiments are run in the same environment (when required), we **store the parameters** of the **environment** inside of a **JSON file.**

As a consequence, when we instantiate an object the **Environment** class, the parameters are **read** from a JSON file.

```
{
    "num_classes": 3,
    "bids": [0, 1, 100],
    "prices": [15.5, 30.7, 60.2, 70.6, 90.8],
    "noise_mean": 0.0,
    "noise_std": 5.0,
    "conv_rates": [
        [0.36, 0.77, 0.51, 0.28, 0.12],
        [0.22, 0.27, 0.61, 0.59, 0.35],
        [0.12, 0.16, 0.31, 0.62, 0.59]
    ],
    "class_probabilities": [1.0, 0.0, 0.0]
}
```

```
{
    "num_classes": 3,
    "bids": [0, 1, 100],
    "prices": [15.5, 30.7, 60.2, 70.6, 90.8],
    "noise_mean": 0.0,
    "noise_std": 5.0,
    "conv_rates": [
        [0.36, 0.77, 0.51, 0.28, 0.12],
        [0.22, 0.27, 0.61, 0.59, 0.35],
        [0.12, 0.16, 0.31, 0.62, 0.59]
    ],
    "class_probabilities": [0.34, 0.33, 0.33]
}
```

```
{
    "num_classes": 3,
    "bids": [0, 1, 100],
    "prices": [15.5, 30.7, 60.2, 70.6, 90.8],
    "noise_mean": 0.0,
    "noise_std": 5.0,
    "phase_length": 120,
    "class_probabilities": [1.0, 0.0, 0.0],
    "conv_rates": [
        [[0.36, 0.77, 0.51, 0.28, 0.12],
        [0.22, 0.27, 0.61, 0.59, 0.35],
        [0.12, 0.16, 0.31, 0.62, 0.59]],

        [[0.29, 0.68, 0.41, 0.6, 0.4],
        [0.17, 0.21, 0.53, 0.50, 0.28],
        [0.8, 0.11, 0.25, 0.53, 0.50]],

        [[0.40, 0.81, 0.42, 0.32, 0.14],
        [0.26, 0.35, 0.69, 0.67, 0.40],
        [0.20, 0.18, 0.35, 0.65, 0.62]]
    ],
    "horizon": 365
}
```

Step 1: Learning for pricing

In this scenario:

- All users belong to a **single class** (let's call it *C1*)
- Information about the **advertising** part of the problem is **known**
- Information about the **pricing** part is **lacking**

To tackle this scenario and learn online information about the pricing part, we will apply two algorithms: **UCB1** and **Thompson Sampling**.

Multi-Armed Bandit Problem

Imagine a scenario in which you have multiple slot machines to choose from, each with an unknown probability of giving a **reward**. The slot machines will be the **arms**.

The **goal** is to minimize the cumulative regret over a series of trials by establishing which one is the best arm to pull.

In our scenario, the arms are the given **price**.

UCB1

Key Idea:
UCB1 balances the exploration-exploitation trade-off, trying to both explore the arms with uncertain rewards and exploit arms that seem promising.

Upper Confidence Bound (UCB):

- For each arm, UCB1 calculates an upper confidence bound representing the uncertainty in the estimated reward
- This upper bound is based on the arm's historical performance and a confidence level parameter

Benefits:

- UCB1 is simple, efficient, and widely used in various applications, such as online advertising, clinical trials, and recommendation systems
- It strikes a balance between exploration and exploitation, leading to good overall performance

UCB1

Considerations:

- The choice of the exploration level parameter impacts the algorithm's behaviour. A smaller value encourages exploitation, while a larger value favours exploration
- UCB1 assumes that the underlying reward distribution remains stationary, which may not hold in some dynamic environments.

Performance:

UCB1 has been theoretically analyzed and shown to achieve near-optimal regret bounds, making it a strong choice for solving the multi-armed bandit problem.

Thompson Sampling (TS)

Key Idea:
Thompson Sampling takes a Bayesian approach to the problem, modeling the uncertainty about the reward probabilities for each arm.

Probability Distributions:

- For each arm, Thompson Sampling maintains a probability distribution (usually a **Beta** distribution) over possible reward probabilities.
- These distributions represent our beliefs about the arms' performance.

Benefits:

- Thompson Sampling is effective in situations where the reward probabilities can change over time or exhibit complex patterns
- It adapts quickly to emerging trends and uncertainties in the environment

Thompson Sampling (TS)

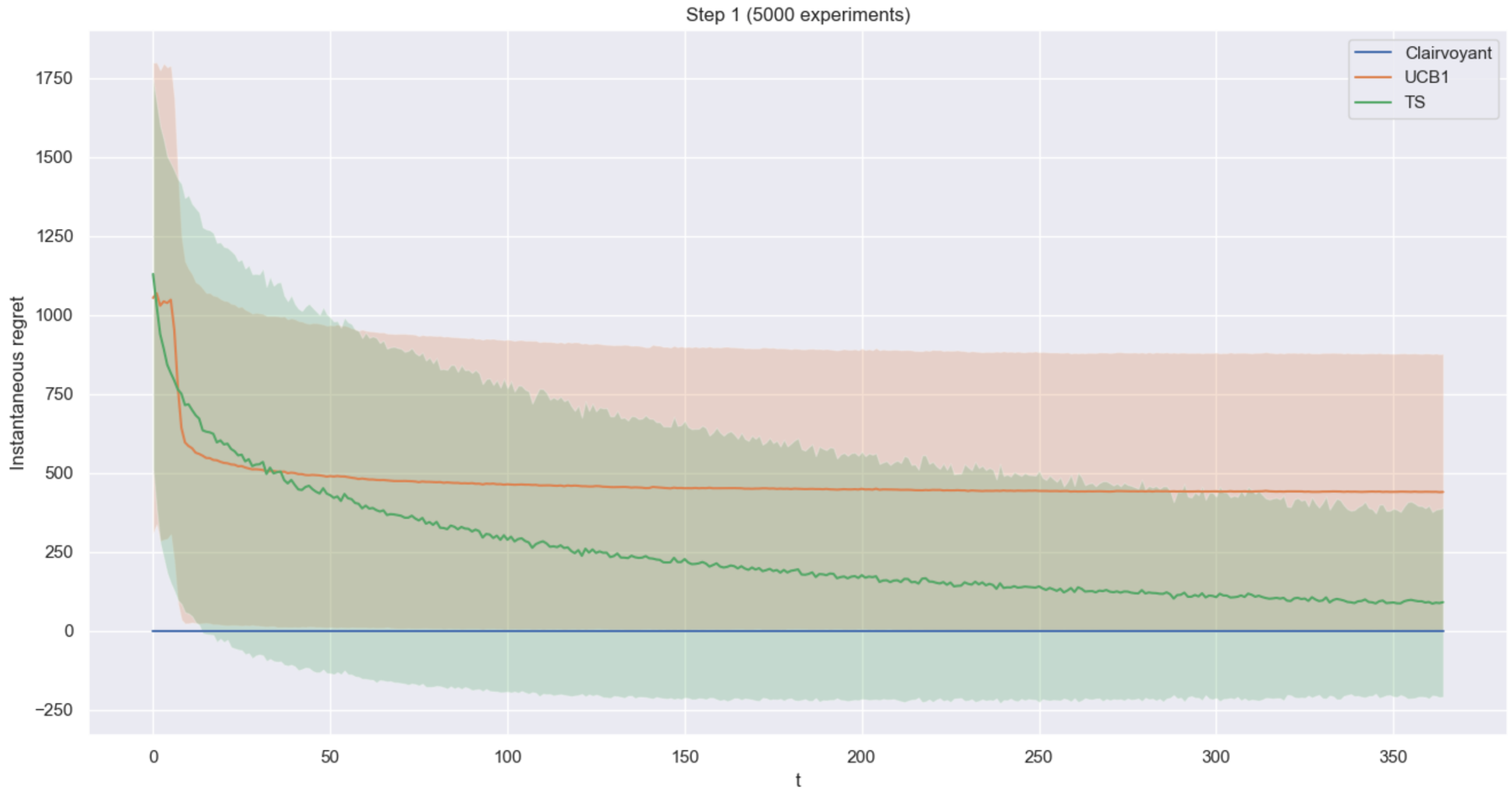
Considerations:

- The choice of probability distribution to model arm rewards can affect the algorithm's performance
- Thompson Sampling requires maintaining and updating probability distributions, which may be computationally intensive for a large number of arms

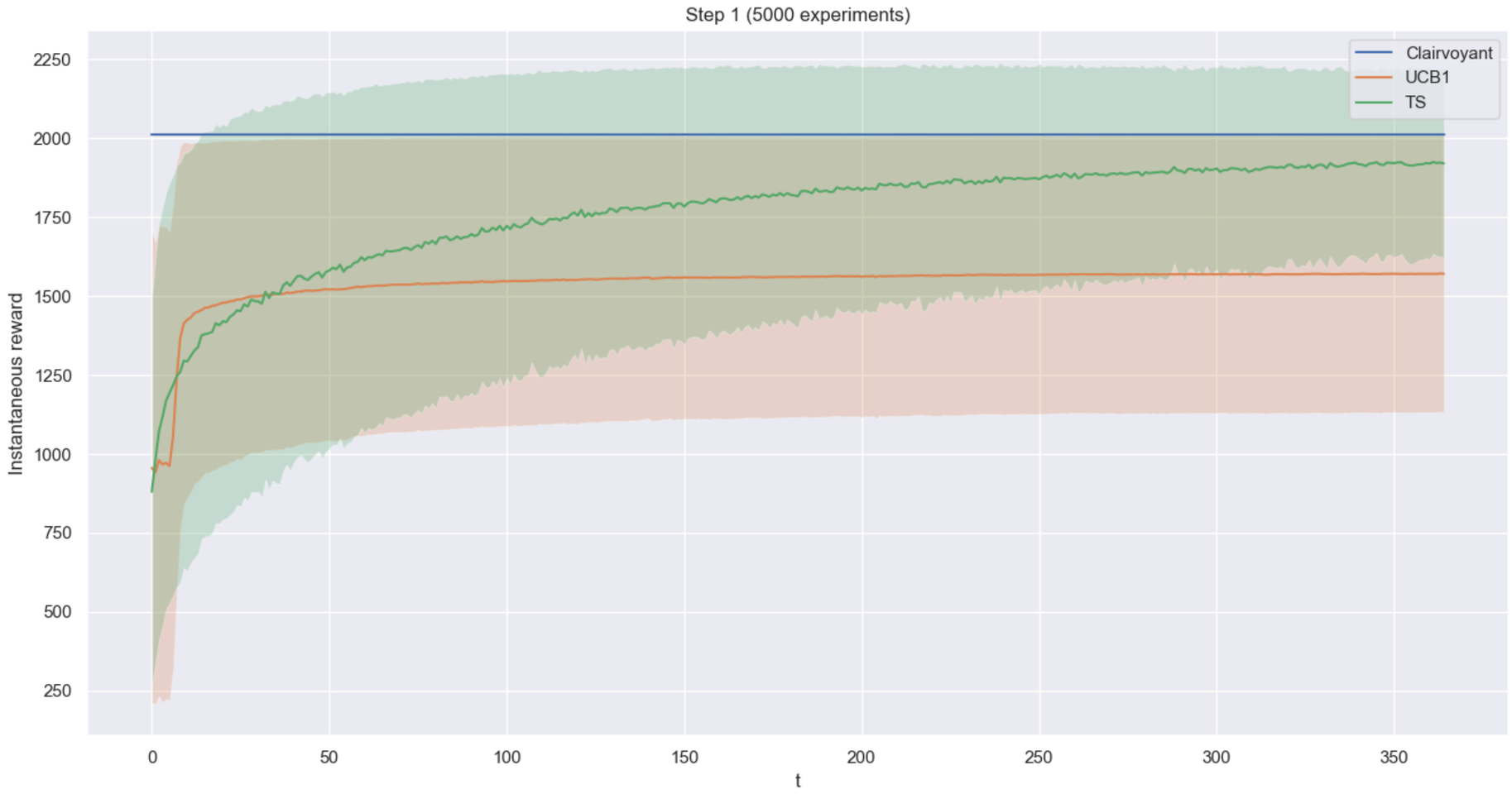
Performance:

Thompson Sampling has been proven to achieve near-optimal regret bounds in various scenarios. It usually performs much better than the plan UCB1, but its assumptions (i.e. the need to have a specific probability distribution over the arms) limit its practical applications. Nonetheless, it is one of the best algorithms to use in the context of online pricing.

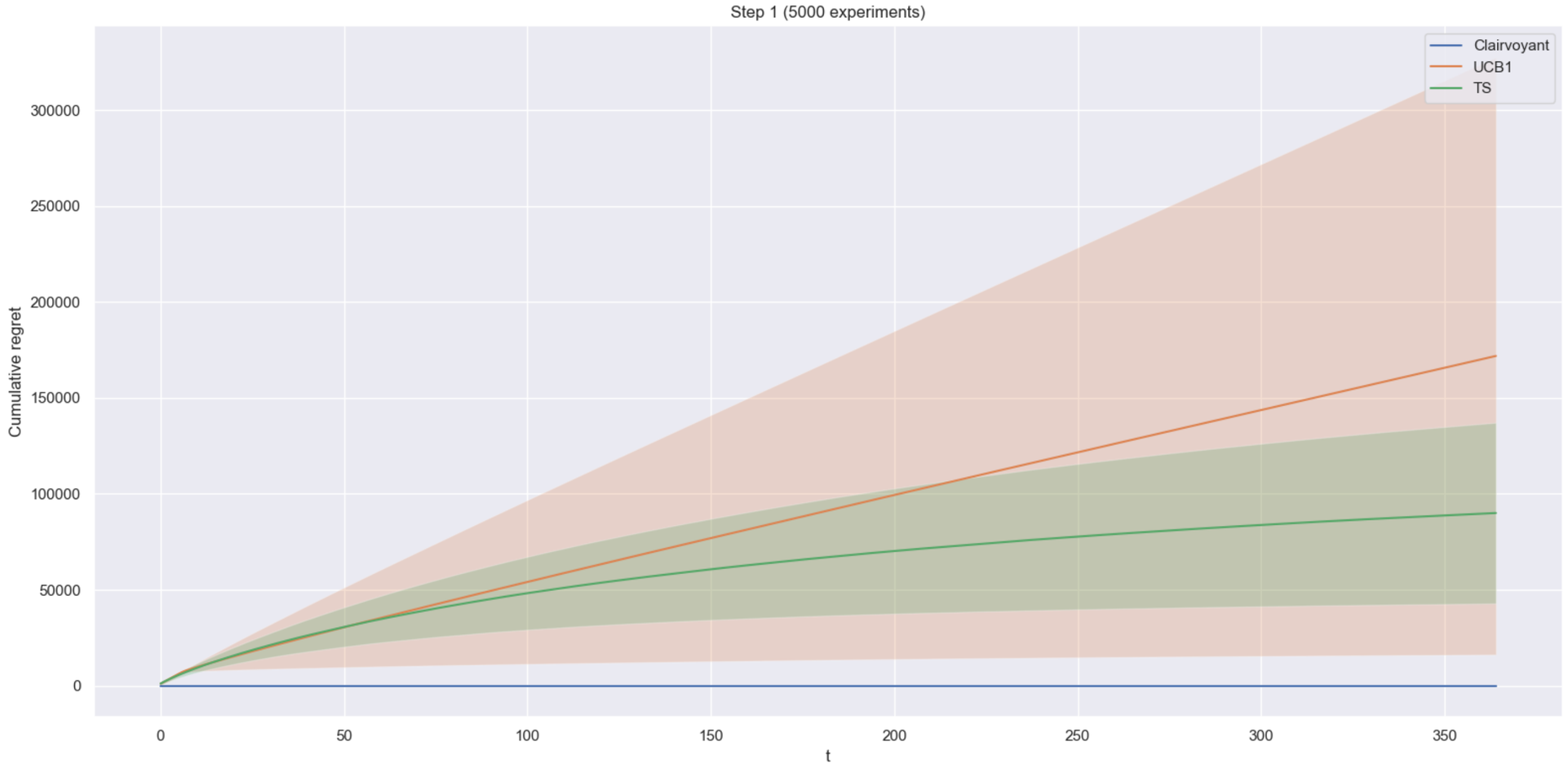
Instantaneous Regret



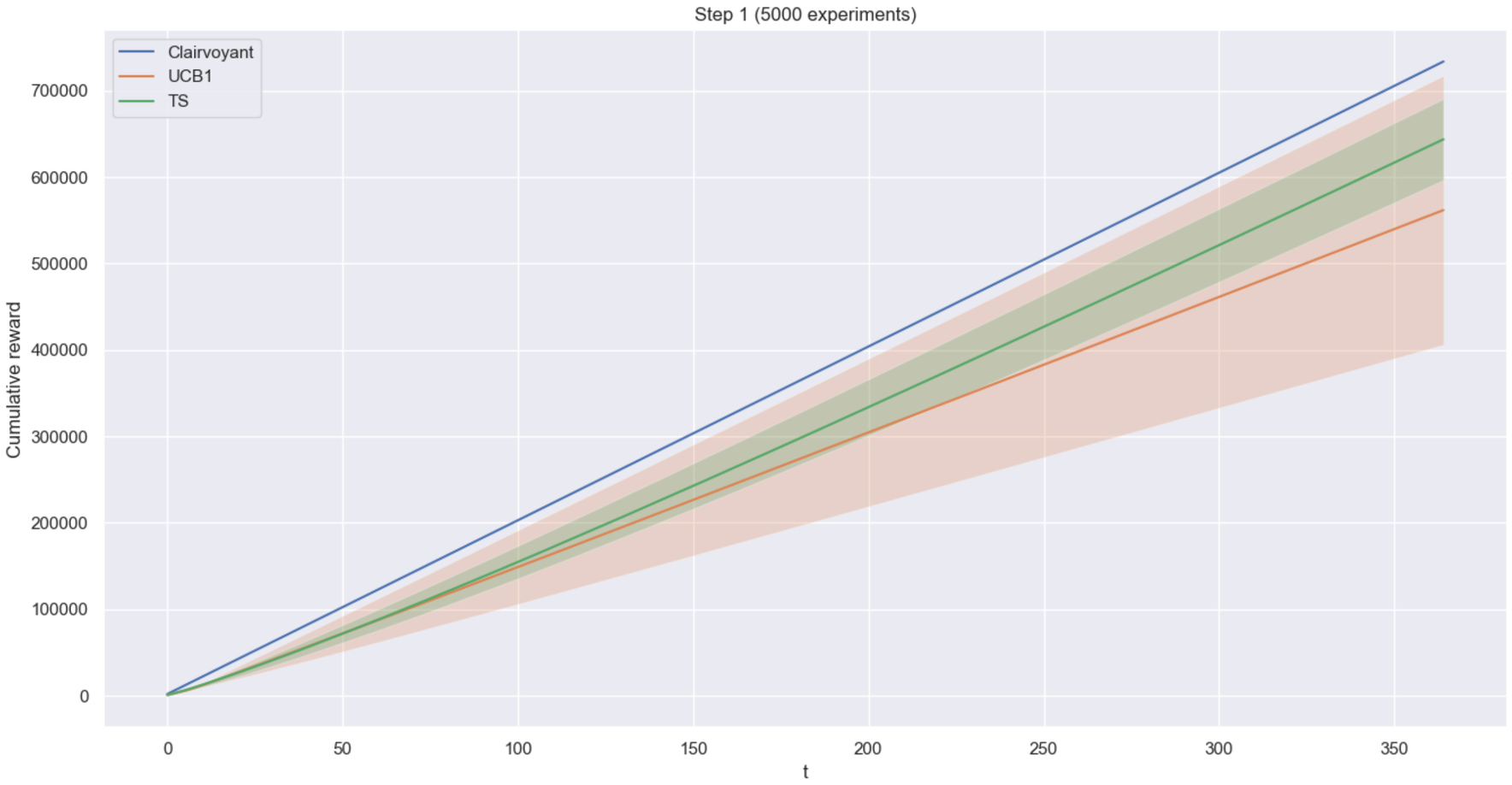
Instantaneous Reward



Cumulative Regret



Cumulative Reward



Comments and Conclusions

Both algorithms are able to achieve a **logarithmic cumulative regret**. In particular:

- **TS** is able to almost match the optimal solution and its approximation becomes better at every time step, as we can see from the plot of the instantaneous regret
- **UCB1** converges on average further w.r.t. the optimal reward. This is expected since it is known that TS performs better than the plain UCB1 in general. As a result, the cumulative regret that UCB1 suffers is higher.

It is noteworthy that the two algorithms, despite showing a very good performance, never reach the **optimal solution**. We think that this is due to the fact that in our scenario UCB1 and TS have to do more than just estimate the conversion rate alone: they have to find out which arm is the best one w.r.t. **the product of the conversion rate they are estimating and its corresponding price**, which is a tougher problem.

Step 2: Learning for advertising

This scenario is the same as Step 1, but this time:

- The **pricing** part is **known**
- The **advertising** part is **not** known

To tackle this scenario, we will apply two algorithms: **GP-UCB** and **GP-TS**, both of which rely on Gaussian Processes (GPs) to model the uncertainty in the advertising curves.

Gaussian Process

GP-UCB and GP-TS utilize Gaussian Processes, a **probabilistic modelling technique**, to represent the uncertainty about the arms' reward distributions.

GPs provide a flexible framework for **modelling complex, unknown functions**, with very mild assumptions.

GP-UCB & GP-TS

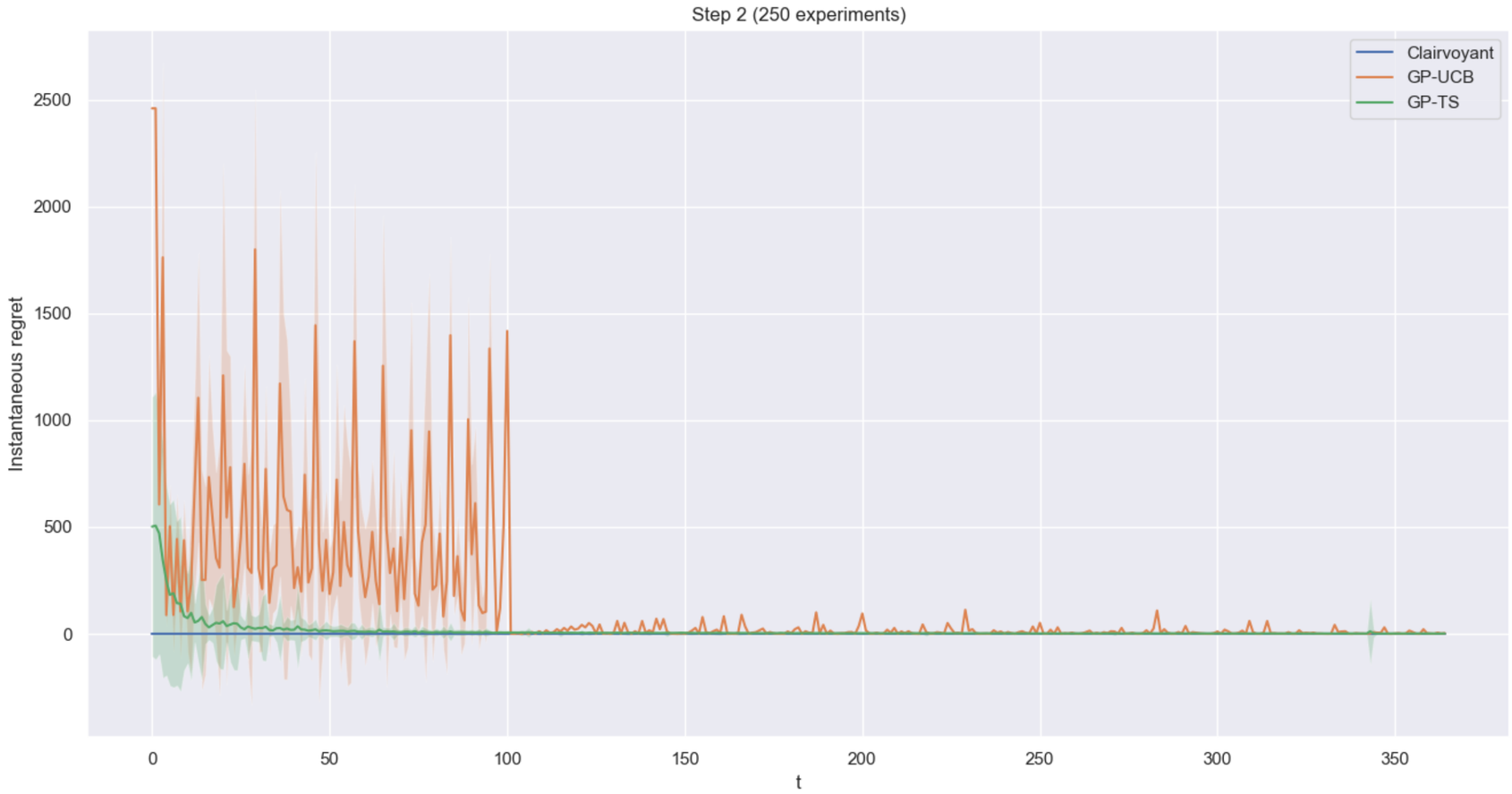
Considerations:

- The choice of GP hyperparameters and the kernel function significantly impact algorithm performance
- GP-MAB assumes that the reward distribution remains stationary, which may not hold in dynamic environments

Performance:

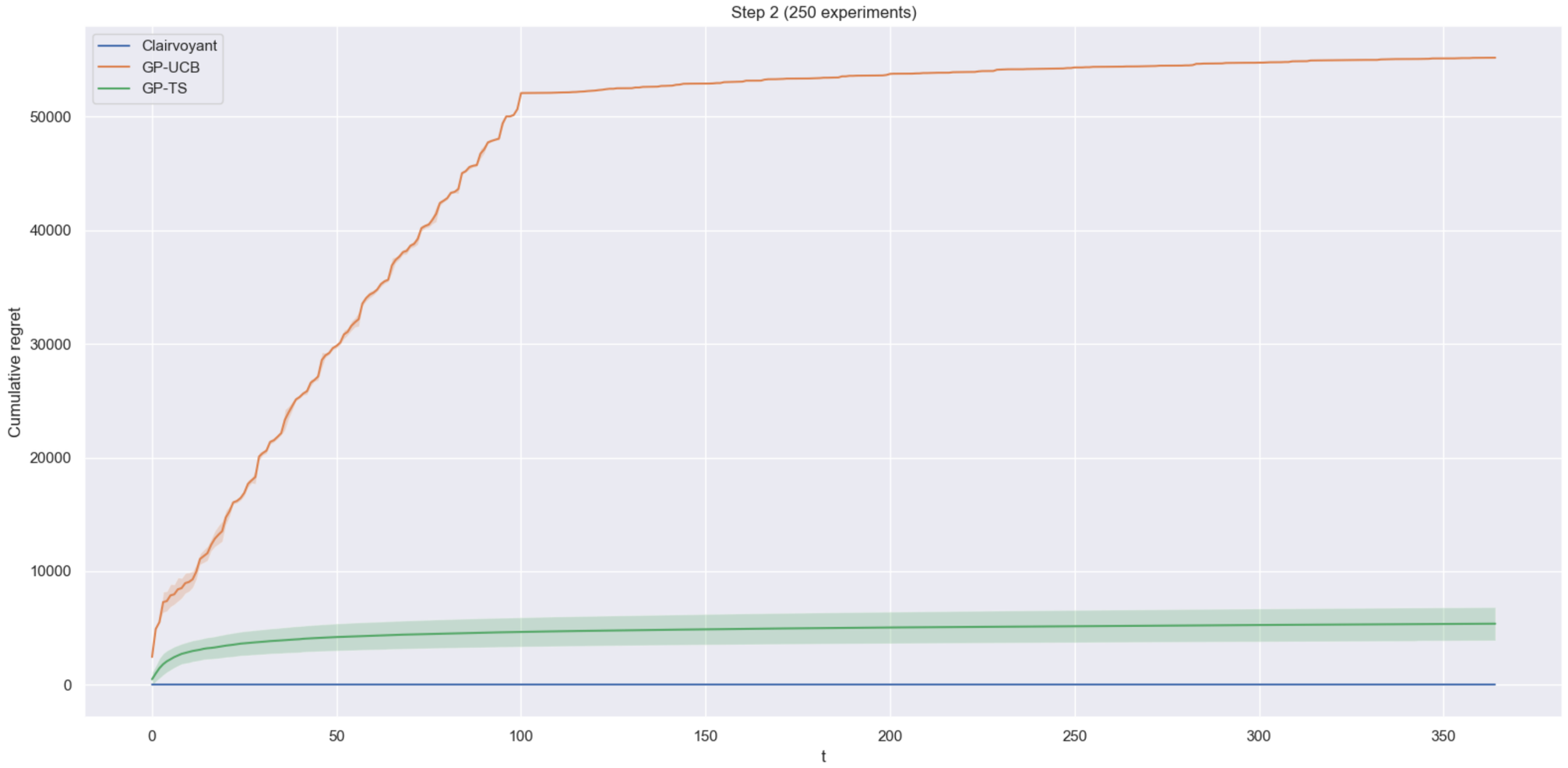
GP-MAB has been widely applied in various domains and has demonstrated strong empirical performance

Instantaneous Regret

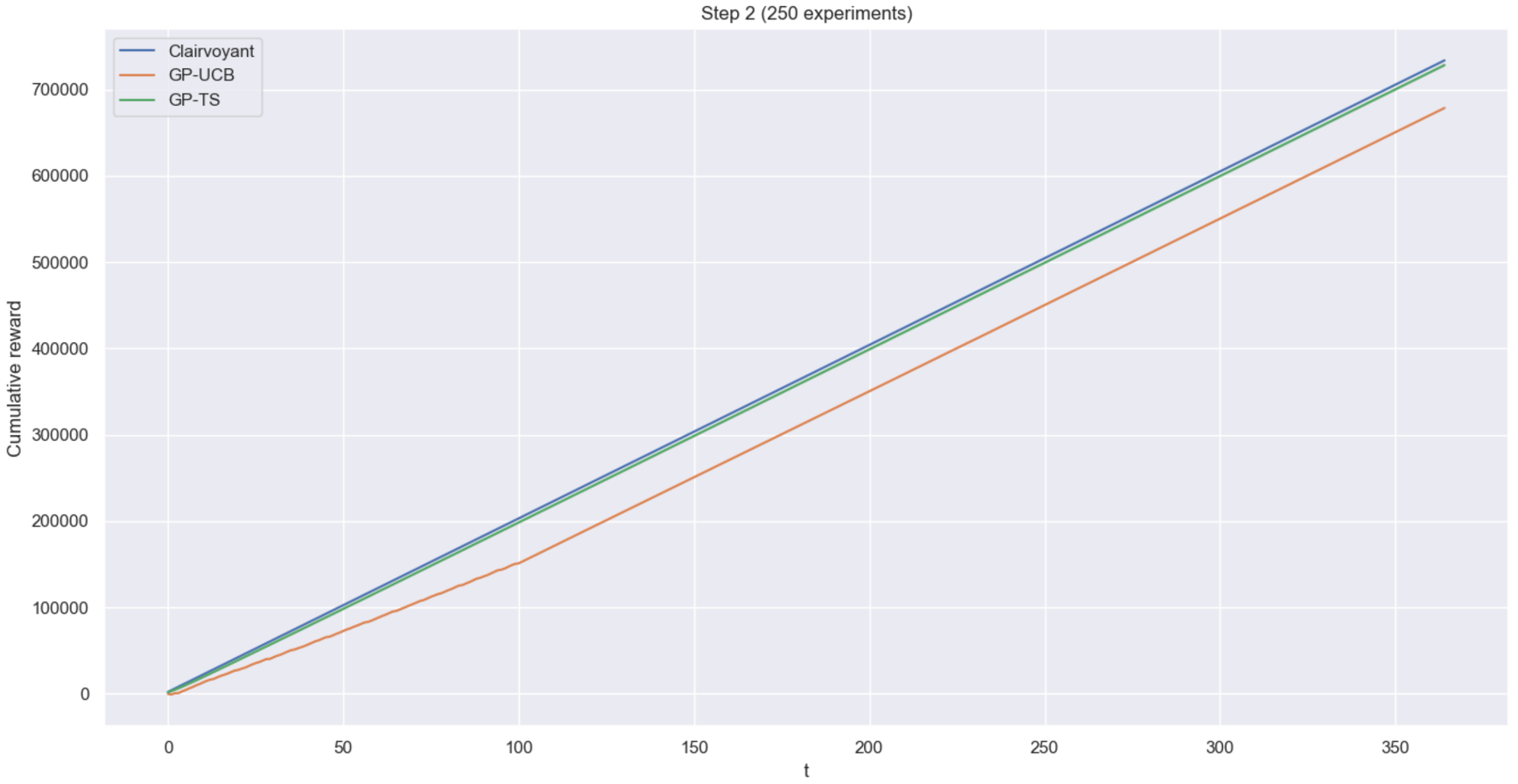


Instantaneous Reward

Cumulative Regret



Cumulative Reward



Comments and Conclusions

Both algorithms are able to achieve a **logarithmic cumulative regret**. In particular:

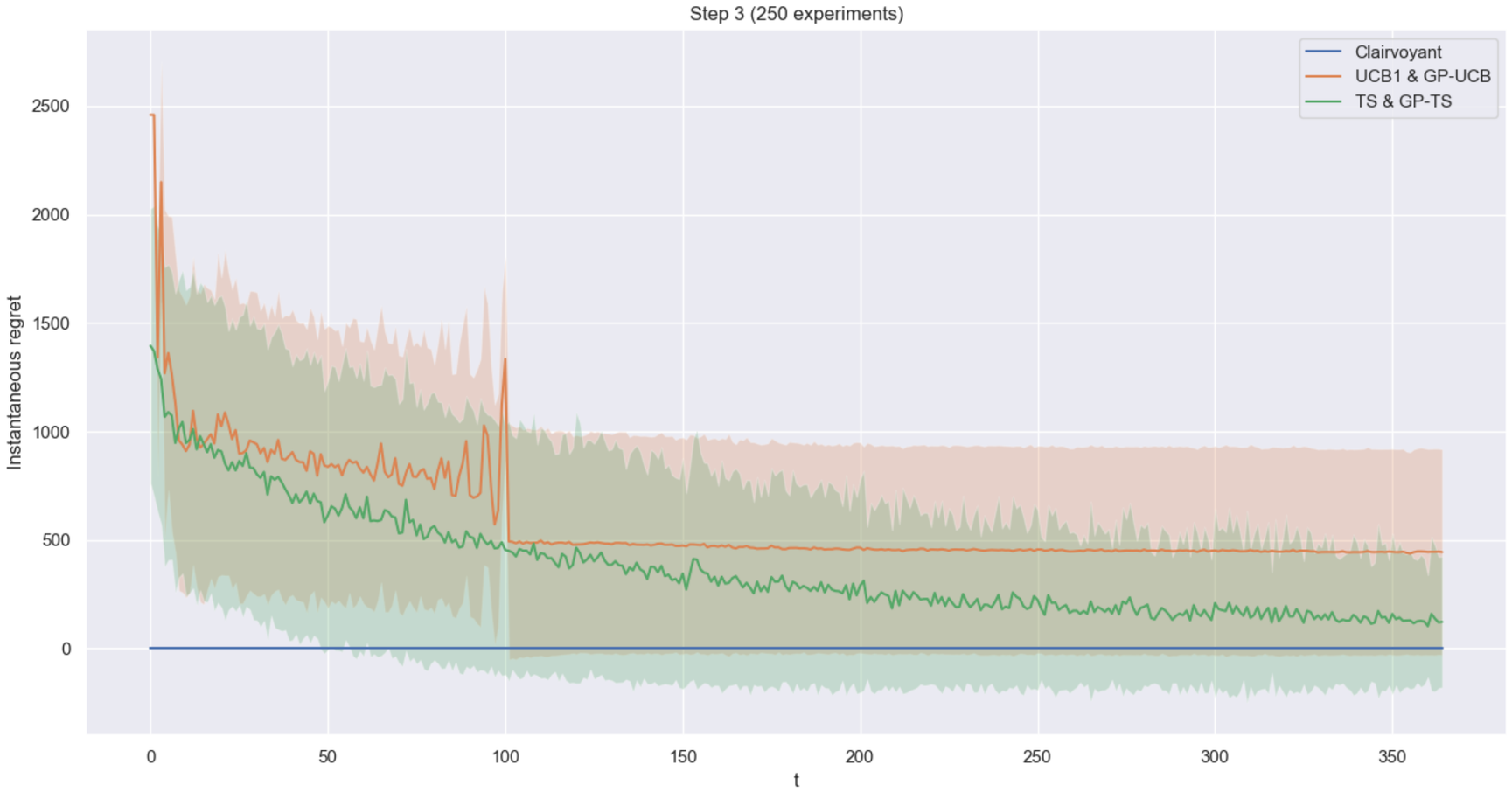
- **GP-TS** shows excellent performance, reaching the optimal solution and stabilizing around it in very few time steps. As a result, the cumulative regret it suffers from is very small.
- **GP-UCB**'s performance is far worse. This is due to the fact that this algorithm requires trying all the arms (in our case, all the bids, which are 100) before starting to optimize the exploration-exploitation tradeoff. As we can see from the plot of the instantaneous regret, this process takes 100 time steps in which the algorithm accumulates a lot of regret. As a result, the regret is basically **linear** for the first 100 time steps and then becomes **logarithmic** shortly after, since the algorithm learned which one is the best arm, as we can see from the plot of cumulative regret.

Step 3: Learning for joint pricing and advertising

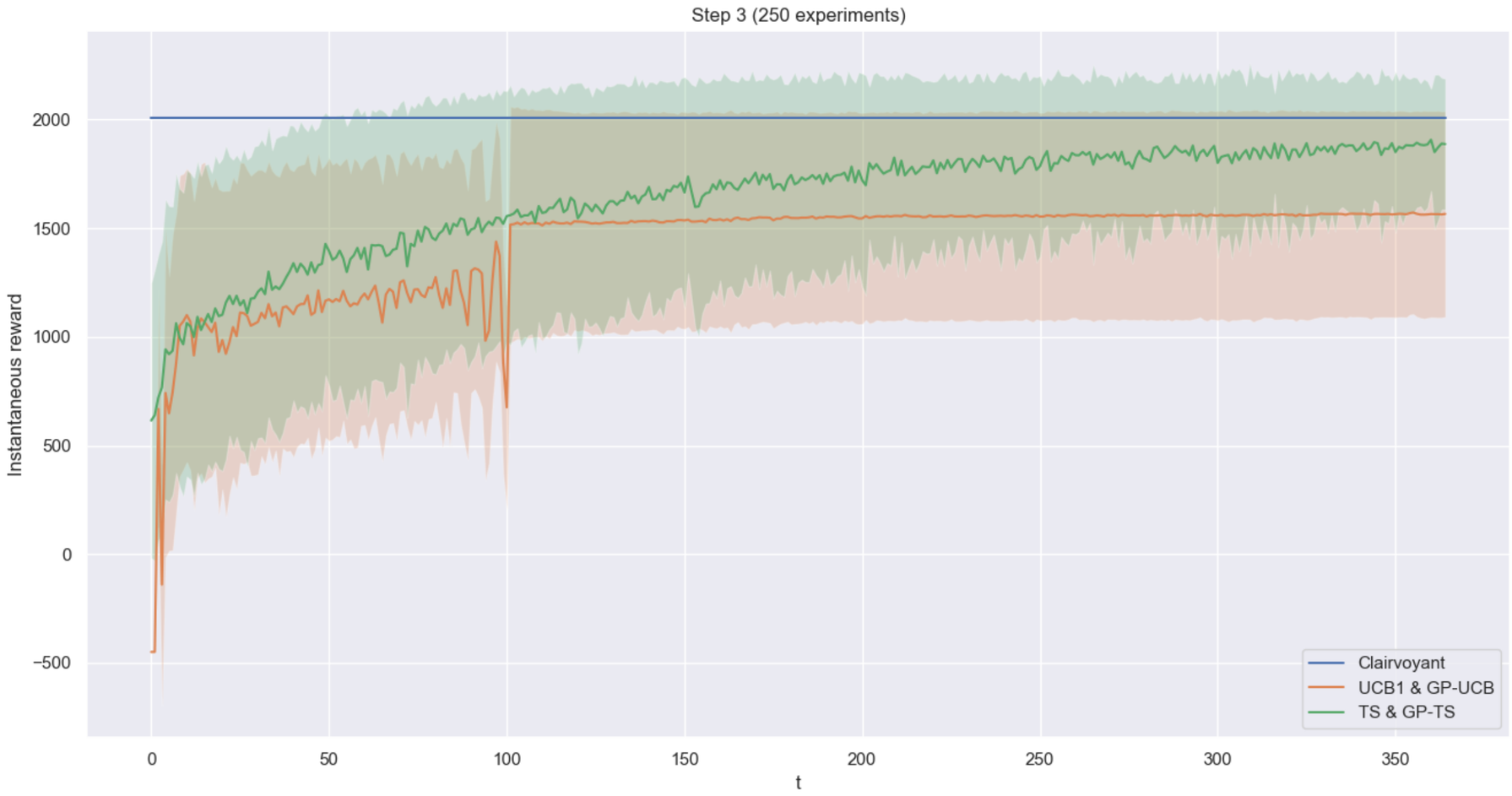
This scenario is the same as Step 1, but this time we have **no prior knowledge** on either the pricing part or the advertising part.

To tackle this scenario, we will **combine** the algorithms of the previous 2 steps. As a result, we will need to run the two algorithms **sequentially**, using the estimate computed by the pricing algorithms to compute the estimate for the advertising algorithms.

Instantaneous Regret



Instantaneous Reward



Cumulative Regret

Cumulative Reward

Comments and Conclusions

Both algorithms are able to achieve a **logarithmic cumulative regret.** The results of this step **strictly depend** on the results of Step 1 and Step 2. We can notice that:

- The combination of TS and GP-TS is the one that achieves the best performance. This is a consequence of the fact that TS and GP-TS are the **best-performing** algorithms in Step 1 and Step 2 respectively.
- No algorithm is able to converge exactly to the **optimal solution**. This is a consequence of the fact that no algorithm was capable of doing so in Step 1
- The regret (both instantaneous and cumulative) of the combination of UCB1 and GP-UCB changes greatly after 100 time steps. This is a consequence of the fact that in Step 2, GP-UCB needs to try all the advertising arms once before looking for the best one.
- All regrets are higher w.r.t. the previous steps. This is expected since now the algorithms need to learn two correlated values at the same time.

Step 4: Contexts and their generation

In a complex situation, we have three different groups of users **(C1, C2, and C3)**, and we **lack** any prior information about how advertising and pricing strategies should be applied to them. We are exploring two scenarios:

1. **Scenario 1:** we already know the structure of the user contexts so we understand how different users are grouped or categorized beforehand.
2. **Scenario 2:** In the second scenario, we do **not have prior** knowledge of the context structure. We need to **learn** this structure from the available data. It's important to note that in this scenario, **we don't even know how many user contexts exist.**

To address both scenarios, we will use two algorithms: GP-UCB and GP-TS. Additionally in Scenario 2, we will pair these algorithms with a context generation algorithm.

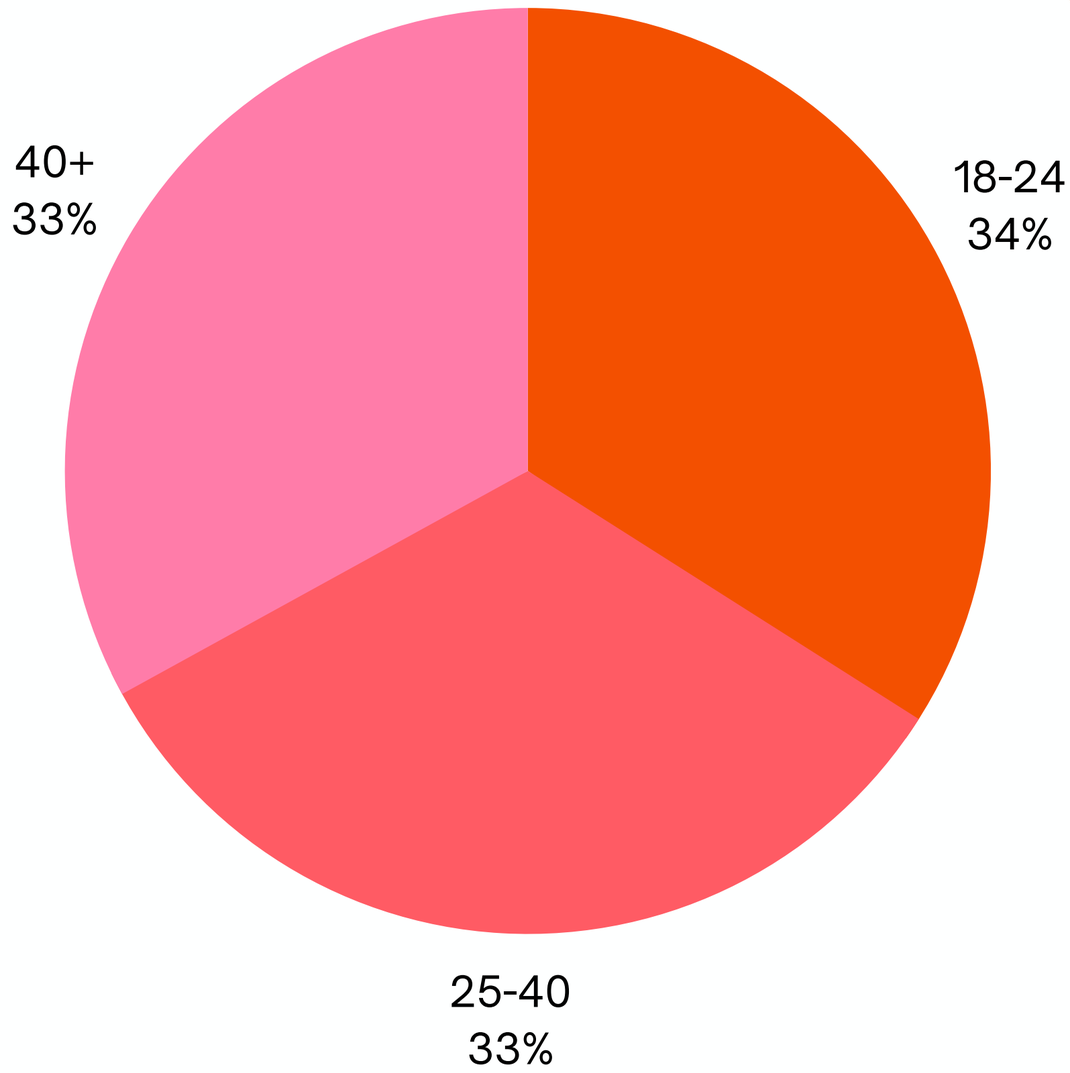
At the end, we will also run GP-UCB and GP-TS algorithms without context generation. We'll treat all users as if they belong to a single context for the entire duration of the analysis.

Classes

01 18-24

02 25-40

03 40+



Context generation algorithm

Real-time context is the information available at the moment of decision-making.
In some cases, it **may not be readily** accessible or is **too costly** to obtain.

The Offline Context Generation algorithm **bridges the gap** by creating context data in advance.
It generates context features based on **historical data, patterns, or known factors relevant to the decision task**.

Considerations:

- The quality and relevance of the generated context features play a critical role in the algorithm's effectiveness.
- Regular updates to the offline-generated context may be necessary to adapt to changing conditions.

Split Condition & Hoeffding Bound

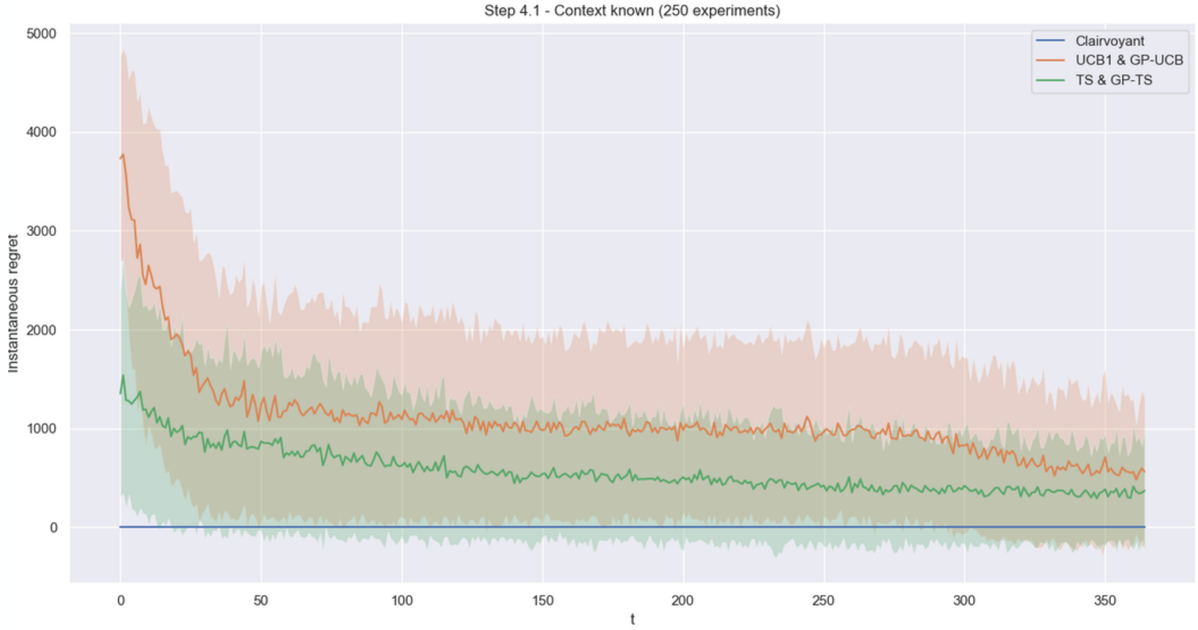
After calculating the value of the **best arms** in each testing context, we select the maximum between the **lower bound weighted sum** of the classes belonging at each context.

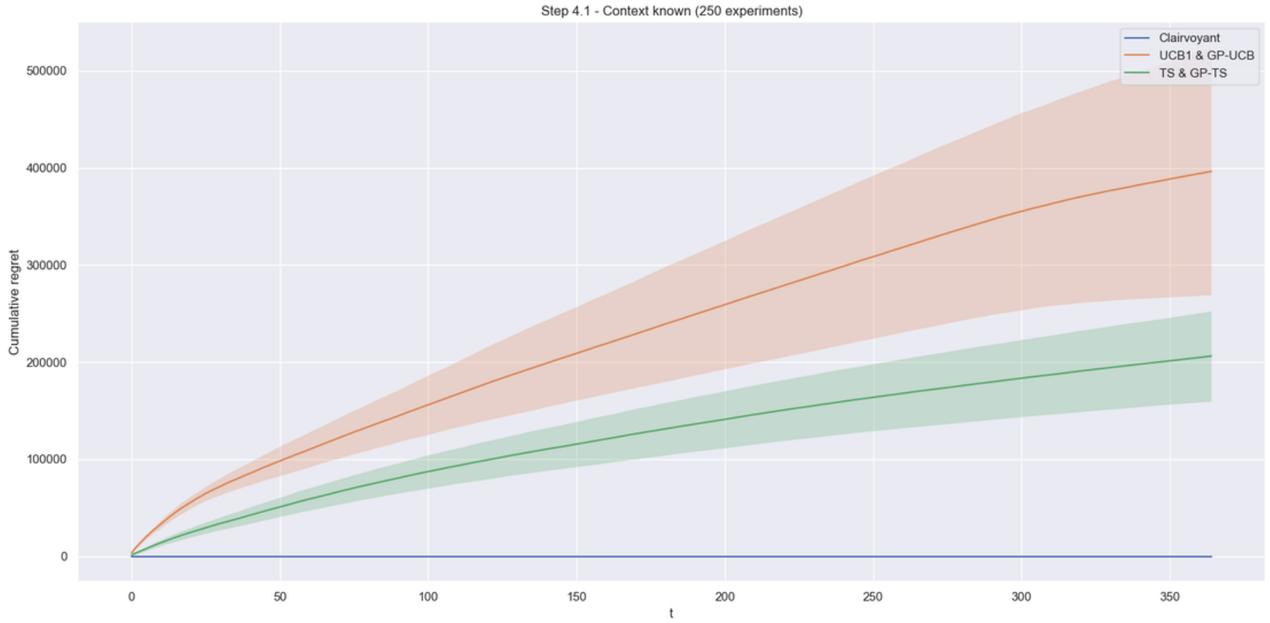
$$\underline{p}_{c_1}\,\underline{\mu}_{a^*_{c_1},c_1} + \underline{p}_{c_2}\,\underline{\mu}_{a^*_{c_2},c_2} \geq \underline{\mu}_{a^*_{c_0},c_0}$$

The **Hoeffding Bound** is a useful tool for estimating lower bounds on probabilities when working with limited sample data. It provides a statistically sound method to calculate lower bounds with a specified level of confidence, making it valuable in various data-driven applications.

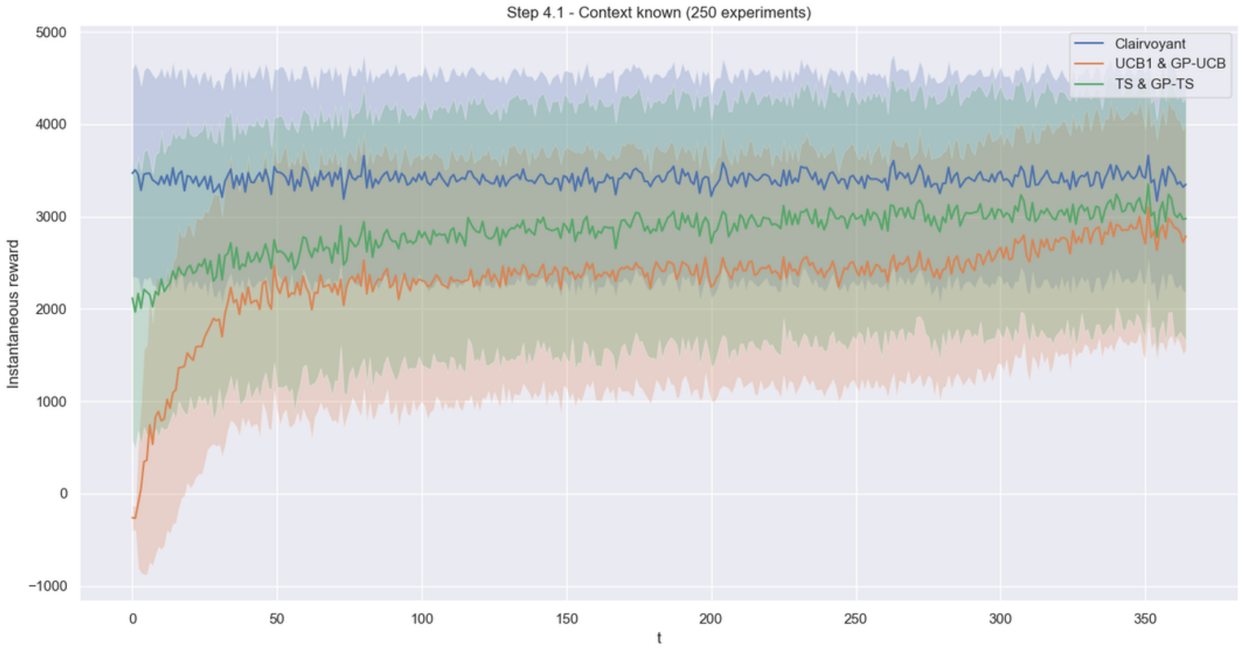
$$\bar{x} - \sqrt{-\frac{\log(\delta)}{2|Z|}}$$

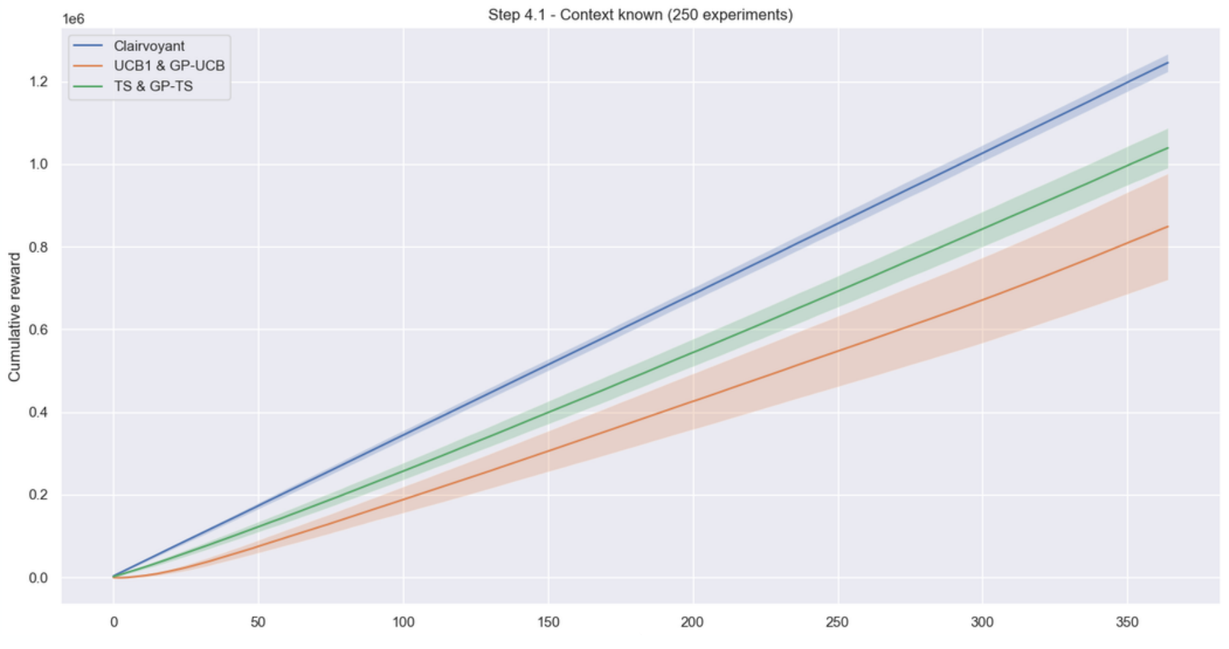
4.1 - Regret


Step 4.1 - Context known (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous regret
t
5000
4000
3000
2000
1000
0
0
50
100
150
200
250
300
350


Step 4.1 - Context known (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
500000
400000
300000
200000
100000
0
0
50
100
150
200
250
300
350

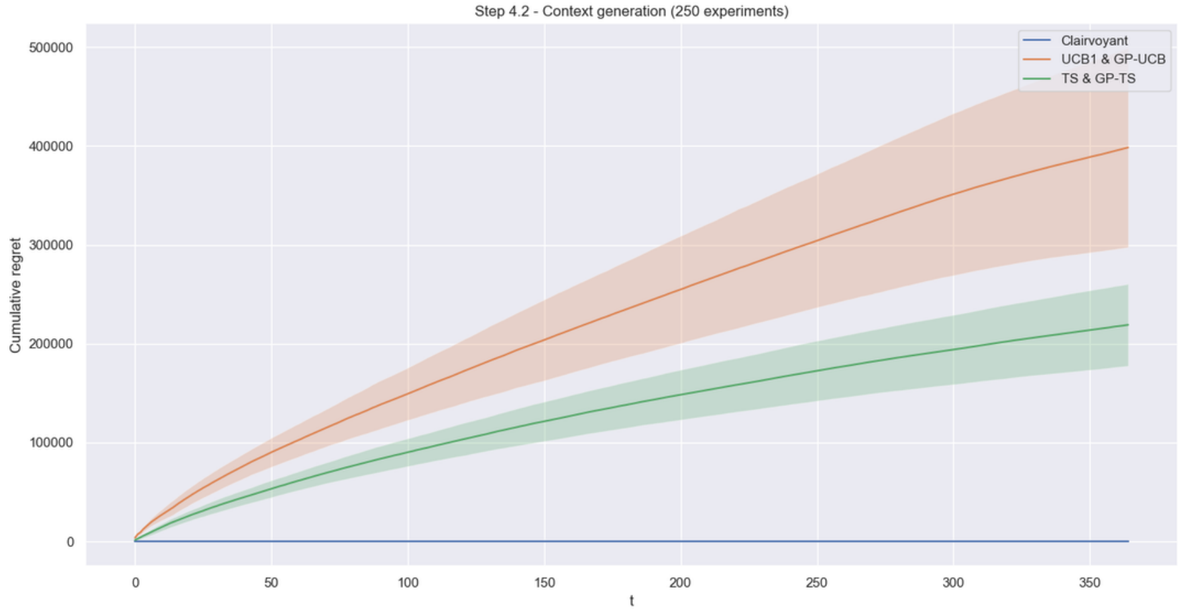
4.1 - Reward


Step 4.1 - Context known (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous reward
t
5000
4000
3000
2000
1000
0
−1000
0
50
100
150
200
250
300
350


Step 4.1 - Context known (250 experiments)
1e6
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative reward
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350

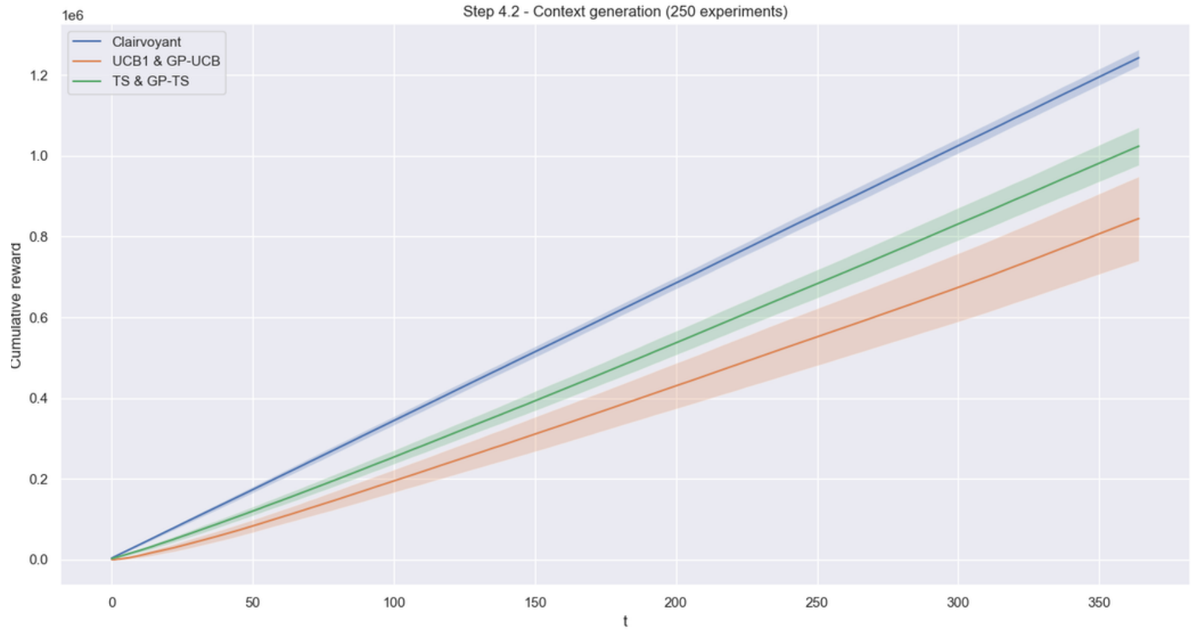
4.2 - Regret


Step 4.2 - Context generation (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous regret
t
0
1000
2000
3000
4000
0
50
100
150
200
250
300
350

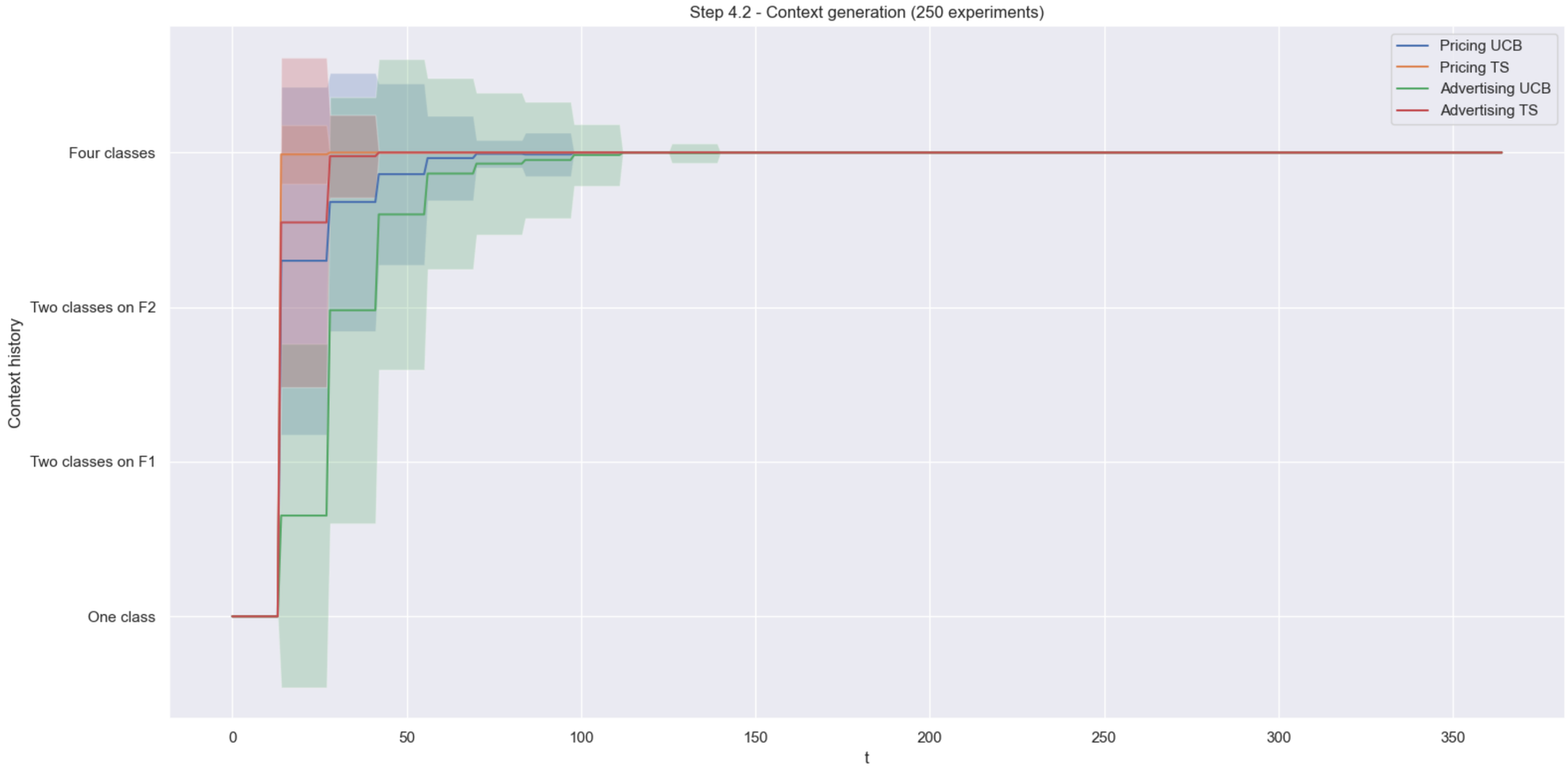

Step 4.2 - Context generation (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
0
100000
200000
300000
400000
500000
0
50
100
150
200
250
300
350

4.2 - Reward


Step 4.2 - Context generation (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous reward
t
4000
3000
2000
1000
0
−1000
0
50
100
150
200
250
300
350

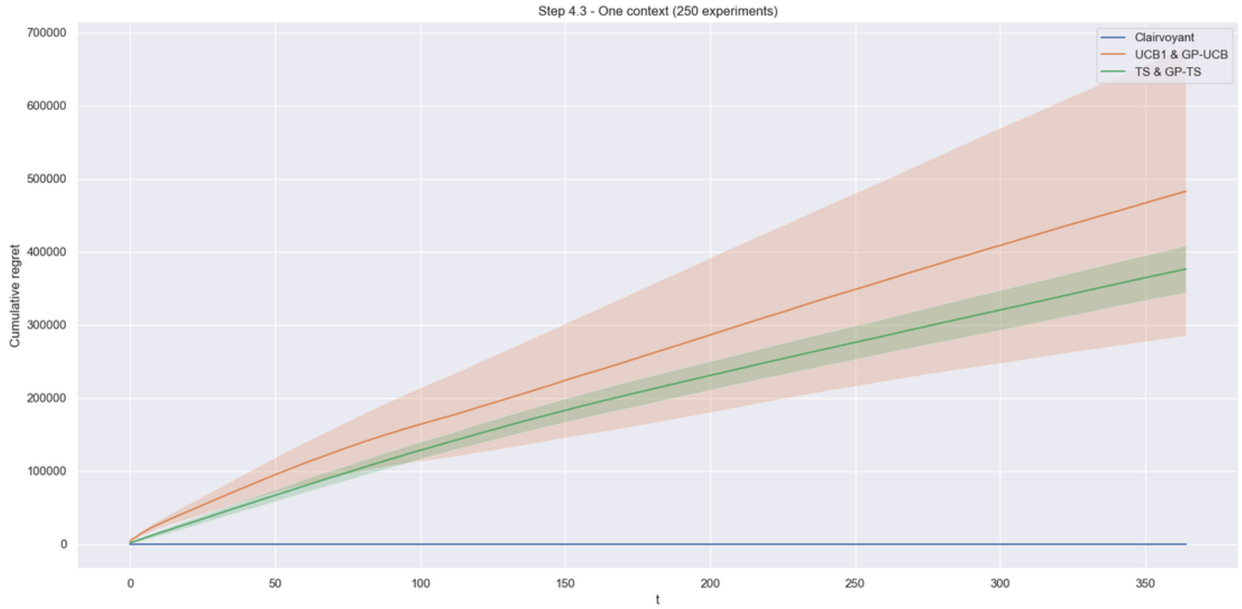

Step 4.2 - Context generation (250 experiments)
1e6
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative reward
t
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350

4.2 - Choice of the Context



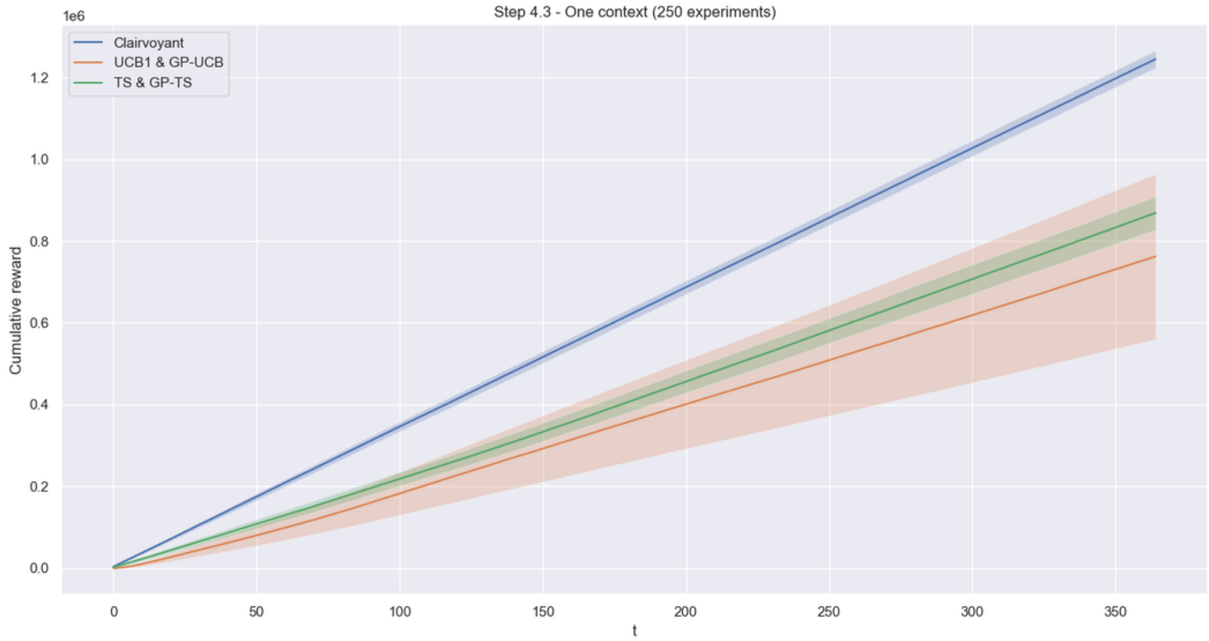
4.3 - Regret


Step 4.3 - One context (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous regret
t
0
1000
2000
3000
4000
0
50
100
150
200
250
300
350

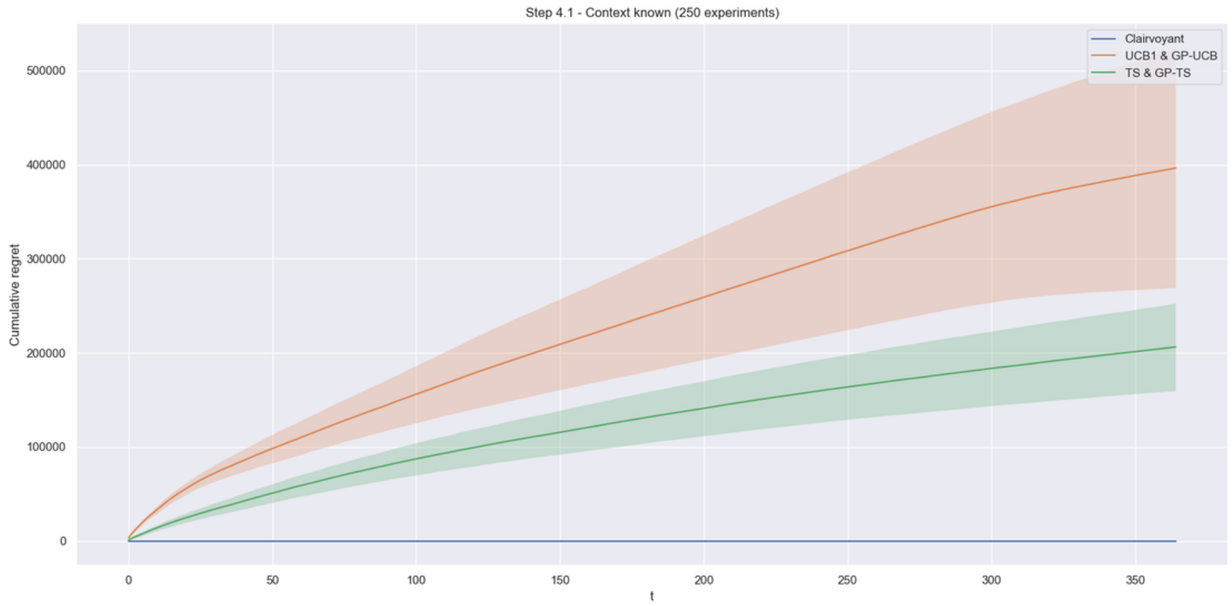

Step 4.3 - One context (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
0
100000
200000
300000
400000
500000
600000
700000
0
50
100
150
200
250
300
350

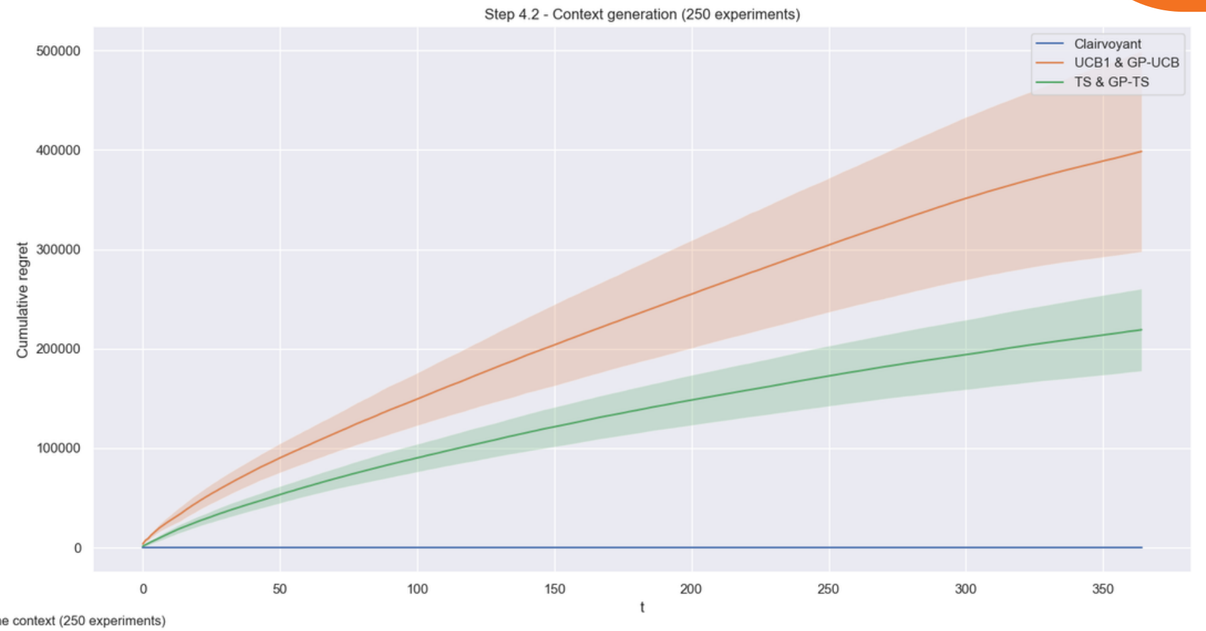
4.3 - Reward

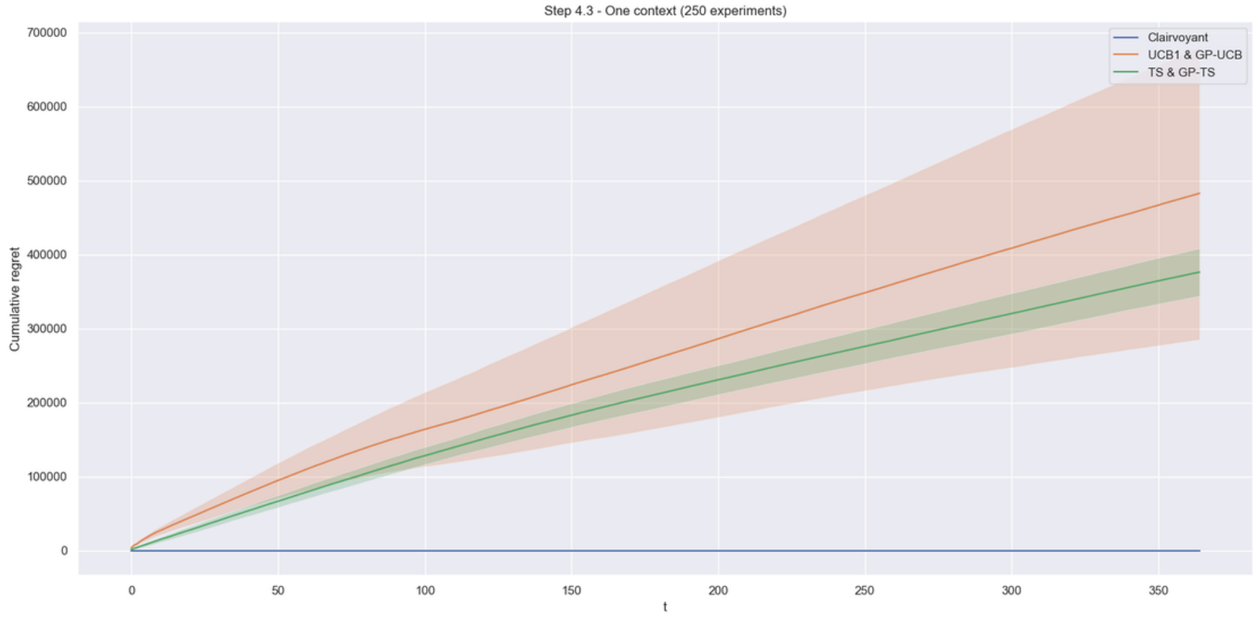

Step 4.3 - One context (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Instantaneous reward
t
5000
4000
3000
2000
1000
0
−1000
0
50
100
150
200
250
300
350


Step 4.3 - One context (250 experiments)
1e6
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative reward
t
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250
300
350

Comparison


Step 4.1 - Context known (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
0
100000
200000
300000
400000
500000
0
50
100
150
200
250
300
350


Step 4.2 - Context generation (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
0
100000
200000
300000
400000
500000
0
50
100
150
200
250
300
350


Step 4.3 - One context (250 experiments)
Clairvoyant
UCB1 & GP-UCB
TS & GP-TS
Cumulative regret
t
0
100000
200000
300000
400000
500000
600000
700000
0
50
100
150
200
250
300
350

Comments and Conclusions

From the plots of the cumulative regret, we can see that the performance of the 3 scenarios is the one expected:

- Step 4.1 is the one that performs **better.**
- Step 4.2's performance is just slightly lower, meaning that our **context generation algorithm is effective.**
- Step 4.3's performance is much worse than both step 4.1 and step 4.2 since in that case the **context is ignored.**

Furthermore, from the plot of the choice of context, we can see that:

- As expected, pricing algorithms (UCB1 & TS) **switch fastly to a context of 4 classes.** This is because they have only 5 arms to estimate and, therefore, a few samples are enough to have a robust per-class model.
- Advertising algorithms (GP-UCB & GP-TS) **take longer to switch to a context of 4 classes** due to the fact that they have to estimate 100 arms. This holds especially for GP-UCB, which again requires one sample for each arm before being able to improve its estimates.
- All the algorithms eventually converge to a context of 4 classes. This is expected since the actual environment is composed of 3 classes.

Step 5: Dealing with non-stationary environments with two abrupt changes

Again we have a **single-user class** called C1.
We **know** the curves related to **advertising problems**, but we have **no prior information** about the **pricing curve**.
Additionally, the pricing curves are not constant; they change over time in a **non-stationary** manner, following **three distinct seasonal phases** throughout the time period.

The three seasonal phases can be considered the following:

1. **Winter:** In this period people travel less often than during the summer but more than in the spring. As a consequence, prices are **slightly lower** w.r.t. the summer season.
2. **Spring:** In this period people tend to not travel very often thus we expect to have **lower prices.**
3. **Summer:** This is the **holiday period** when people travel often hence prices tend to be **high.**

To address this situation, we will apply the UCB1 algorithm along with two variations of the UCB1 algorithm designed specifically for handling non-stationary data**.**

1. **Passive Non-Stationary UCB1:** using **Sliding Window**
2. **Active Non-Stationary UCB1:** using **Change Detection**

Non-stationary environment
Abrupt changes

Phases	Price1 - 15.50$	Price2 - 30.70$	Price3 - 60.20$	Price4 - 70.60$	Price5 - 90.80$
Winter	0.36	0.58	0.51	0.28	0.12
Spring	0.29	0.32	0.41	0.20	0.08
Summer	0.40	0.67	0.81	0.32	0.14

SW-UCB1

Sliding window is a technique used to deal with **non-stationary environment.**
In this step we use the same UCB algorithm implemented for the previous steps but with the difference that now takes into account only the **last k samples**.
This allows the algorithm to keep track of the **changes** within the environment.

In this case we used a window size of **100 days**, that is approximately every time the seasonality change, in order to obtain better performances. This algorithm doesn’t work well with small windows size since the learner won’t be able to learn the data pattern in an efficient way.

SW-UCB1

1. Play once every arm $a \in A$

2. At t , play an arm a_t with $n_{a_t}(t-1,\tau) = 0$ if any, otherwise play arm a_t such that

$$a_t \leftarrow \arg\max_{a \in A} \left\{ \bar{x}_{a,t.\tau} + \sqrt{\frac{2\ \log(t)}{n_a(t-1,\tau)}} \right\}$$

CD-UCB1

In this case, we implement a **Change Detection algorithm** in order to deal with the abrupt changes of the non-stationary environment. We take the same UCB algorithm used in the previous steps and we apply a **CUSUM algorithm**. CUSUM considers every small positive and negative **variation w.r.t a mean** computed on the first **k rewards.**

If the sum of all positive or negative variations exceeds a **defined bound** the algorithm **clears the mean**, and the algorithm starts the learning phase again. The defined bound is a hyperparameter of the problem and the optimal one was selected after performing some hyperparameter tuning.

CD-UCB1

1. inltialize τ_a=0 for each arm $a \in A$
2. for each t

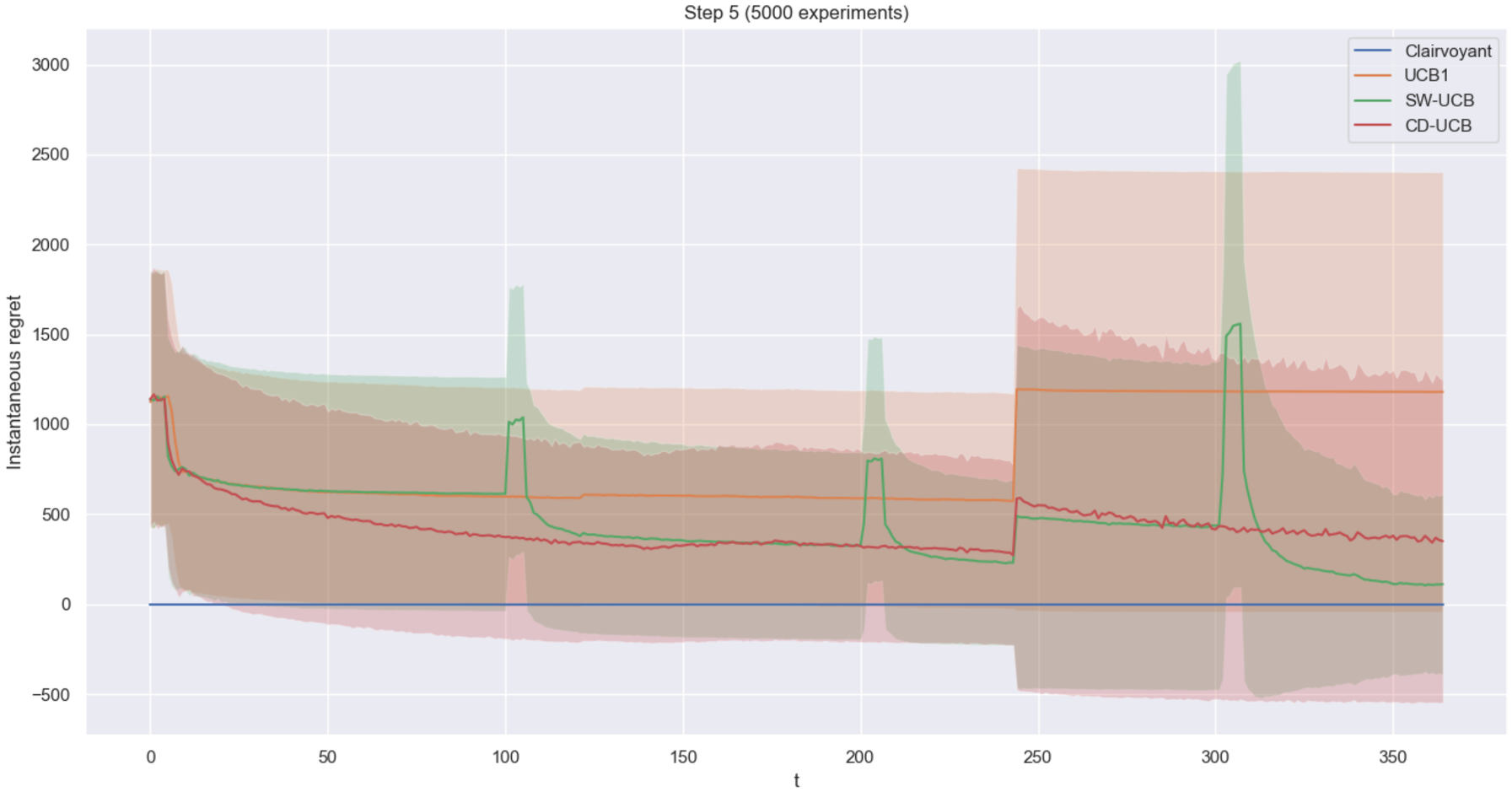
$$a_t \leftarrow \arg\max_{a \in A} \left\{ \bar{x}_{a,\tau_a,t} + \sqrt{\frac{2\log(n(t))}{n_a(\tau_a, t-1)}} \right\} \text{ with probability } 1-\alpha$$

$$a_t \leftarrow \text{random arm with probability } \alpha$$

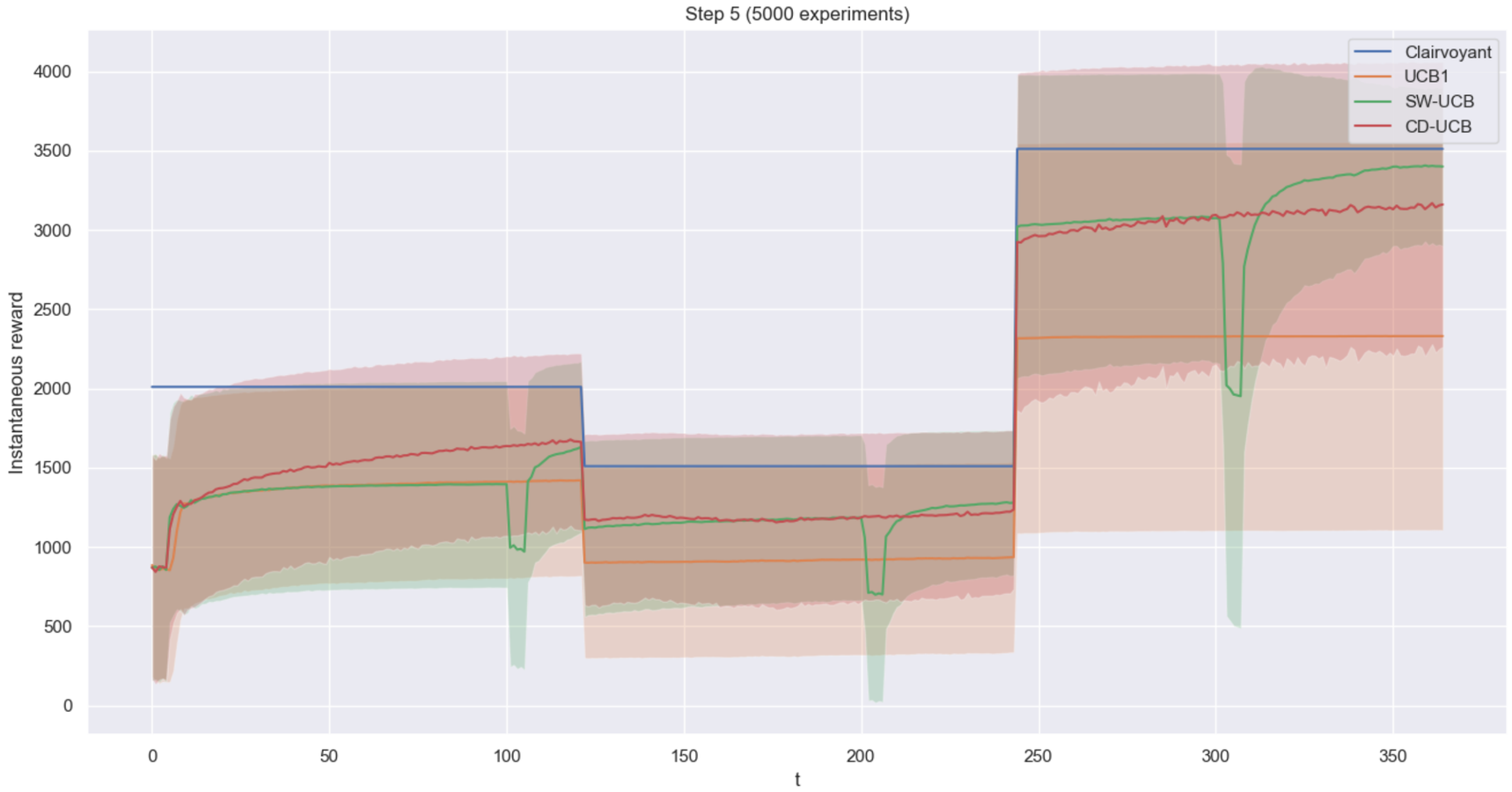
$n(t)$	is the total number of valid samples
$\bar{x}_{a,\tau_a,t}$	is the empirical mean of arm a over the last valid samples
$n_a(\tau_a, t-1)$	is the number of valid samples for arm a

3. collect reward r_t
4. if $CD_a(r_\tau, \ldots, r_t) = 1$ then $\tau_a = t$ and restart CD_a

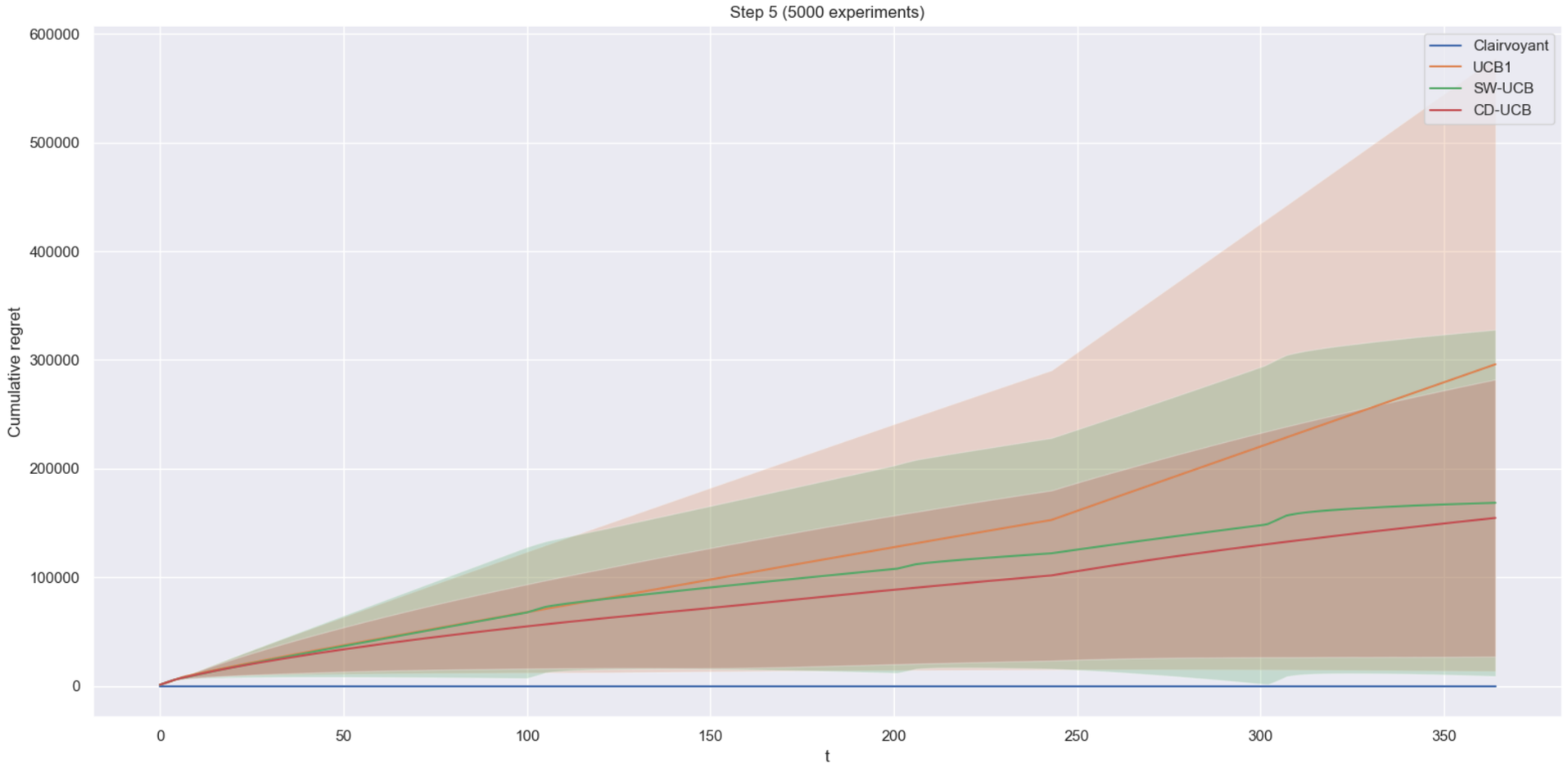
Instantaneous Regret



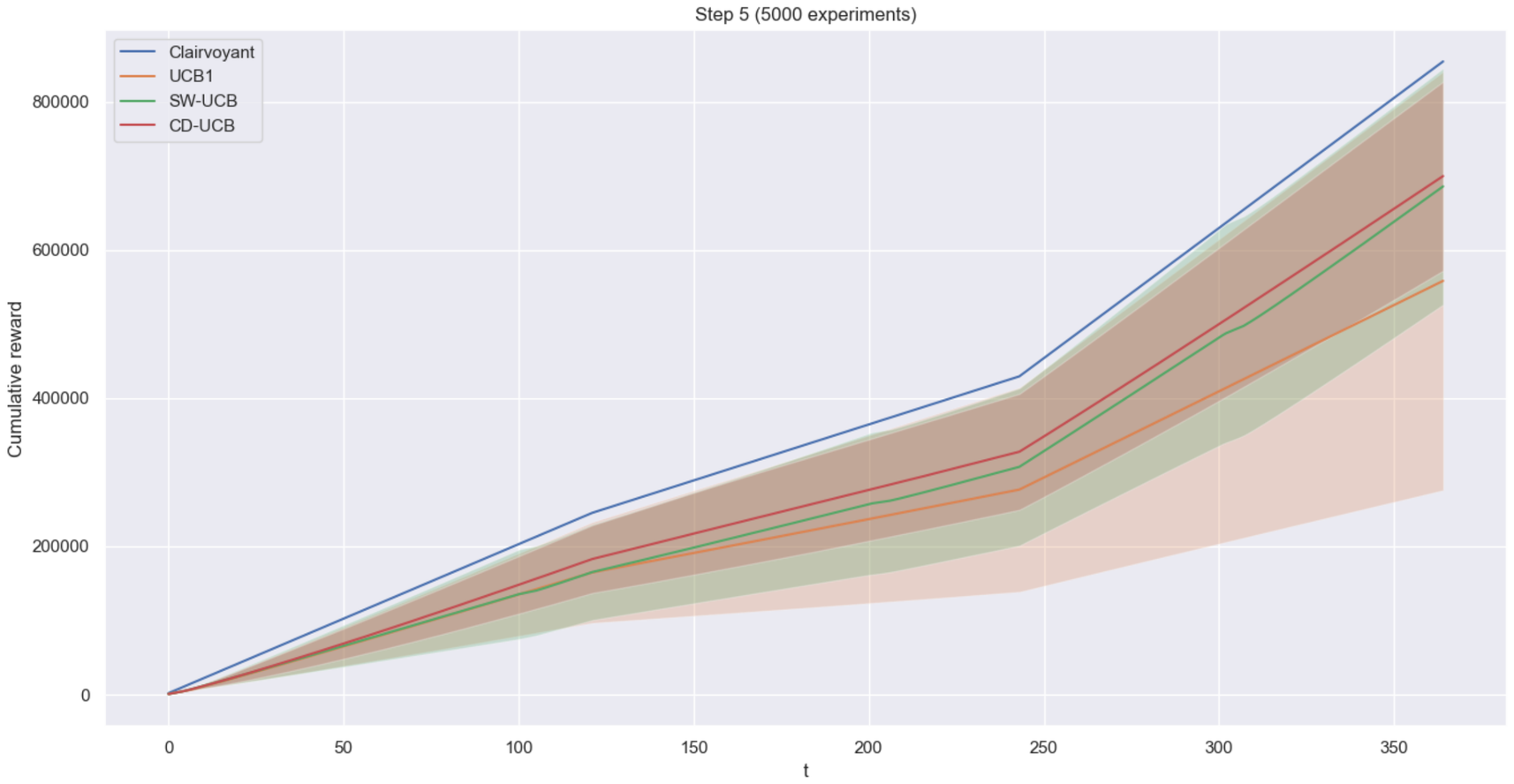
Instantaneous Reward



Cumulative Regret



Cumulative Reward



Comments and Conclusions

What we expect is that in a non-stationary environment, both the Sliding Window UCB1 and the Change Detection UCB1 will outperform the standard UCB1.

As we can see from the plots, in this scenario **UBC1 struggles** to find the optimal price thus it will have a **linear regret**.
On the other hand, both **SW-UCB1 and CD-UCB1** are able to **handle the non-stationary environment** with a regret that is almost **logarithmic** and in particular CD-UCB1 is the one that manages the task in the most efficient way.

We performed a **sensitivity analysis** of the parameters employed in the algorithms and we concluded that Sliding Windows is more efficient when the **window size** is **similar** to the **phase size**, while it strives when the window size is small.
Whereas, for the CUSUM implementation we found the best performances when the exploration value is low and the **threshold value** is about 80.

Step 6: Dealing with non-stationary environments with many abrupt changes

First, in step 6.1, we add the **EXP3 algorithm** in the previous step scenario, considering a **fixed bid**. We expect that EXP3 will perform **worse** than the two non-stationary UCB1.

Next, in step 6.2, we'll consider a different non-stationary scenario characterized by a **higher degree of non-stationarity**. This increased rate of change is modelled by introducing **five phases**, each associated with a different optimal price.
These phases will **rapidly** and **cyclically** change.

In this new scenario, we'll apply the EXP3 algorithm, UCB1, SW-UCB1 and CD-UCB1. We expect that EXP3 will outperform the two non-stationary versions of UCB1.

EXP3

Exponential Weights:

- EXP3 maintains a **list of weights for each arm**, using these weights to decide randomly which action to take next.
- We increase (decrease) the relevant weights when a payoff is good (bad).
- This weighting scheme encourages **exploration** by giving **less-weighted arms a chance to be selected.**

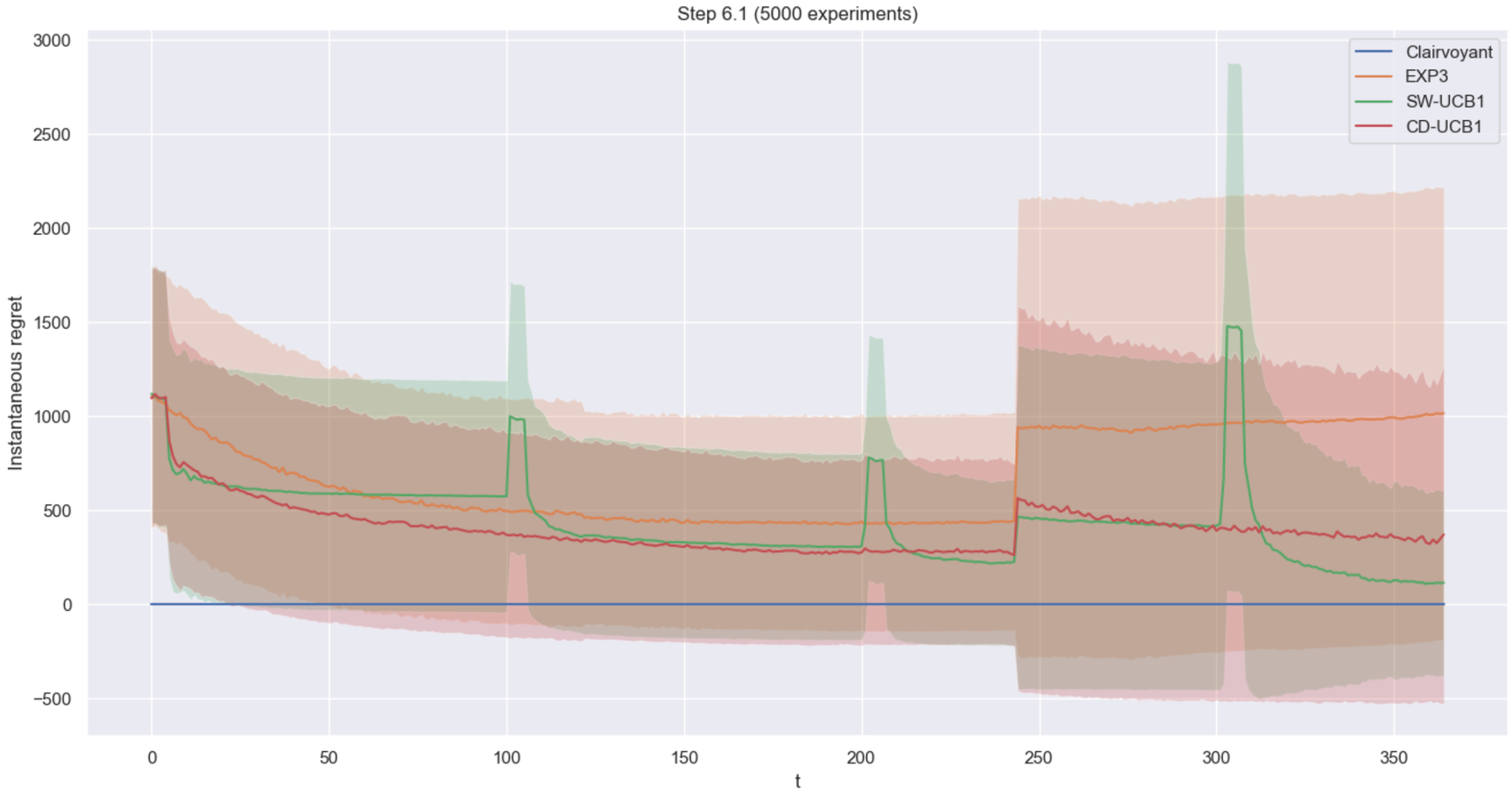
Performance:

EXP3 has been widely studied and has theoretical guarantees for regret bounds, making it a strong choice for solving the multi-armed bandit problem.

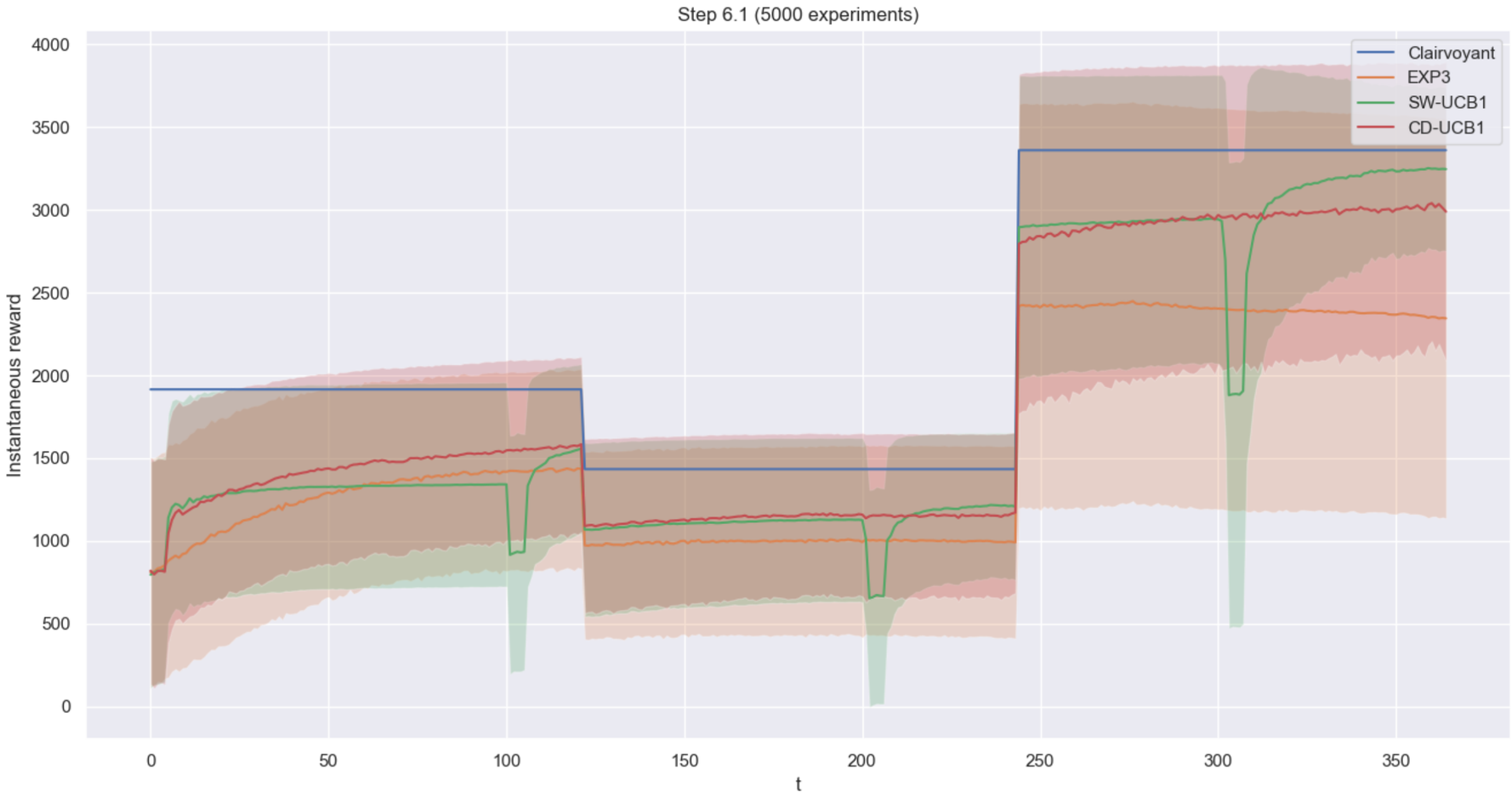
EXP3

1. Given $\gamma \in [0, 1]$, initialize the weights $w_i(1) = 1$ for $i = 1, \ldots, K$.
2. In each round t:
    1. Set $p_i(t) = (1 - \gamma)\dfrac{w_i(t)}{\sum_{j=1}^{K} w_j(t)} + \dfrac{\gamma}{K}$ for each i.
    2. Draw the next action i_t randomly according to the distribution of $p_i(t)$.
    3. Observe reward $x_{i_t}(t)$.
    4. Define the estimated reward $\hat{x}_{i_t}(t)$ to be $x_{i_t}(t)/p_{i_t}(t)$.
    5. Set $w_{i_t}(t + 1) = w_{i_t}(t)e^{\gamma \hat{x}_{i_t}(t)/K}$
    6. Set all other $w_j(t + 1) = w_j(t)$.

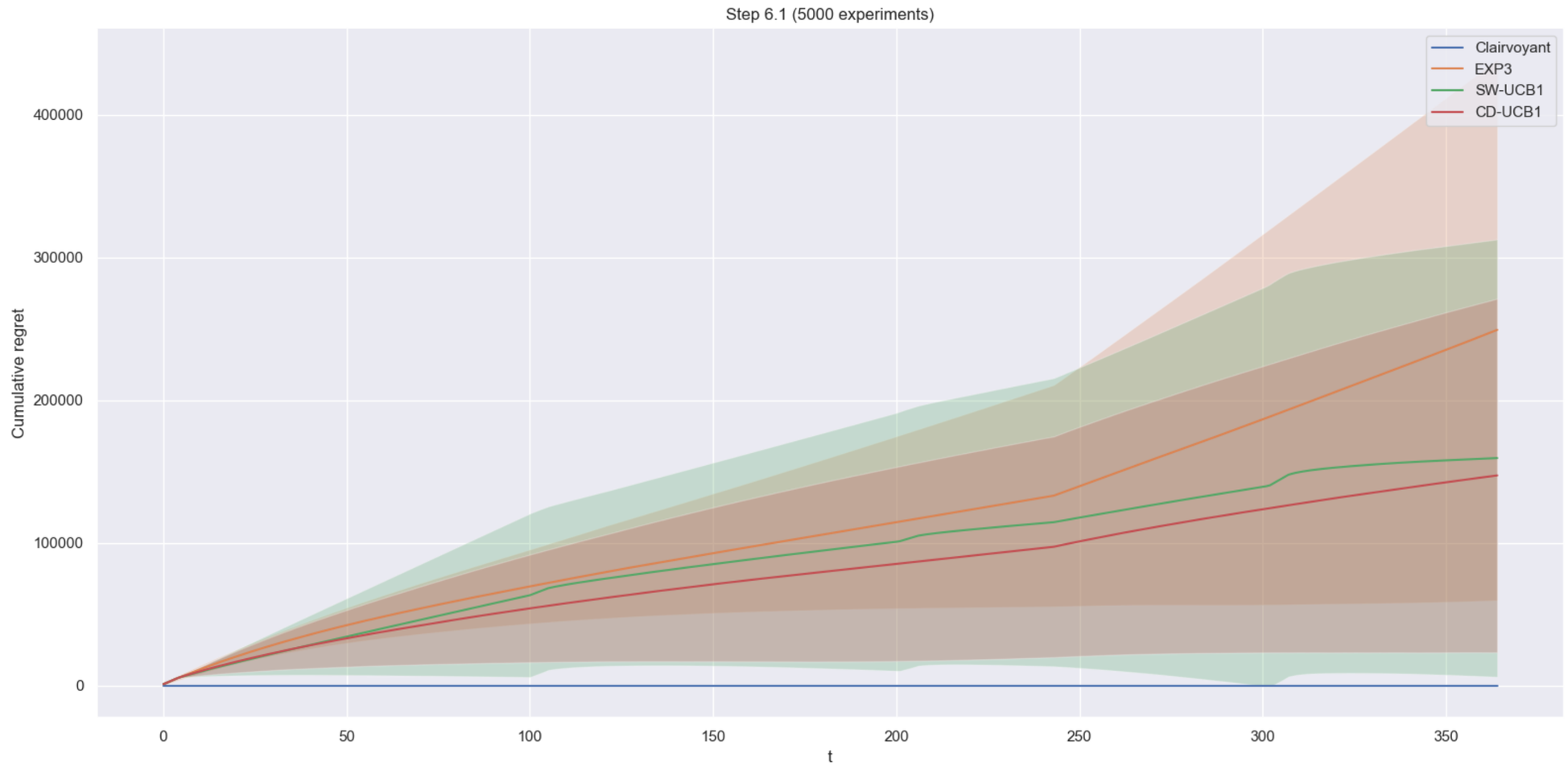
6.1 - Instantaneous Regret



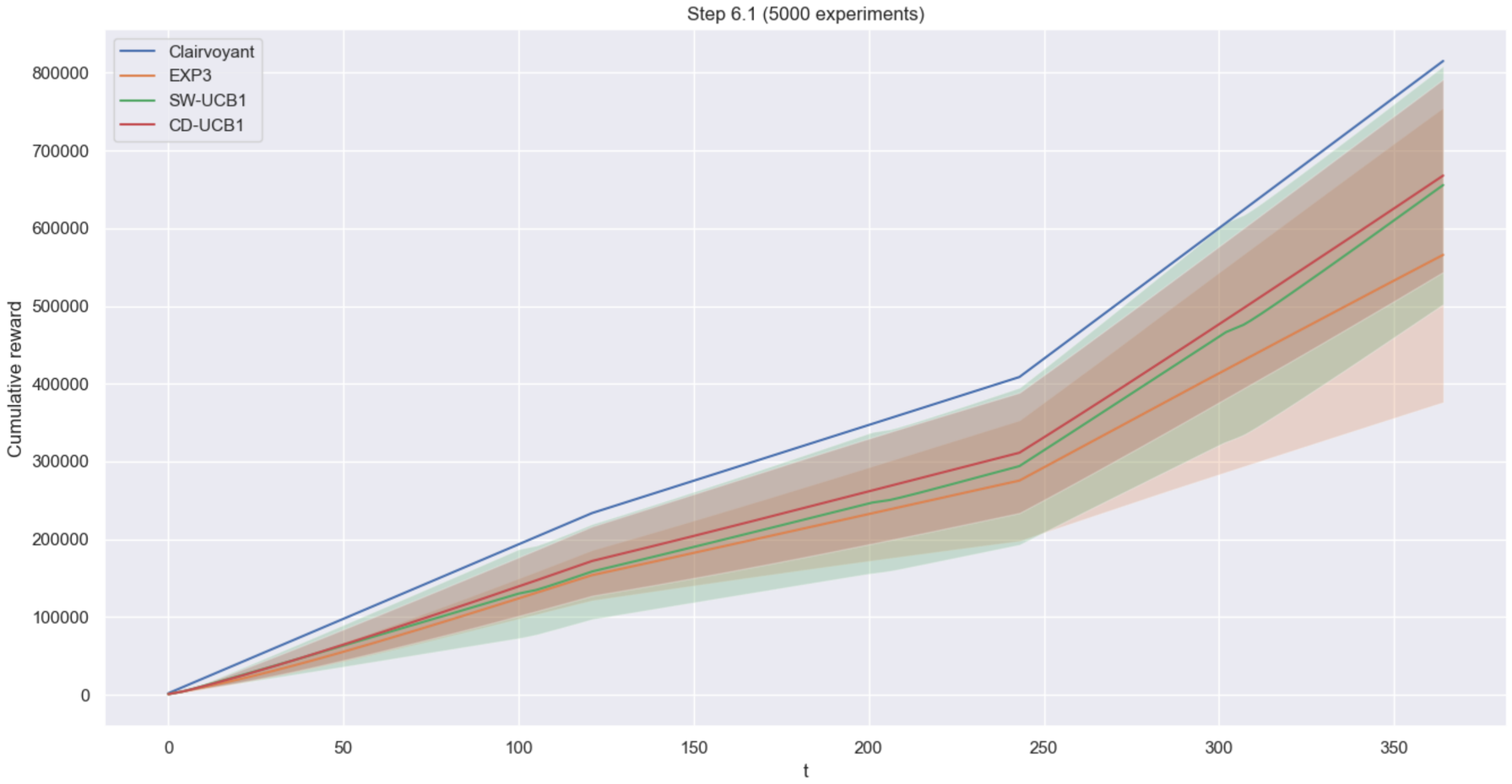
6.1 - Instantaneous Reward



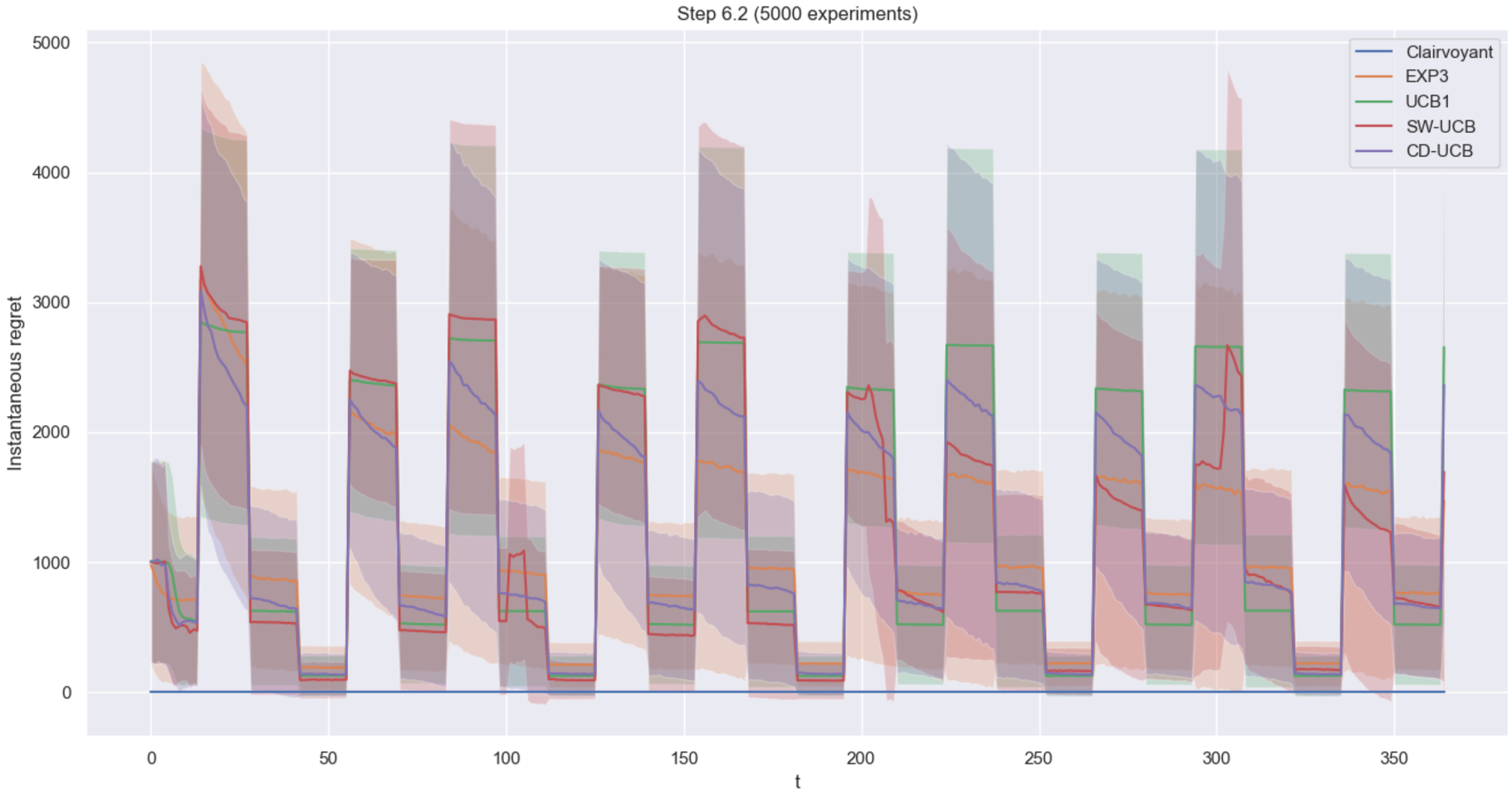
6.1 - Cumulative Regret



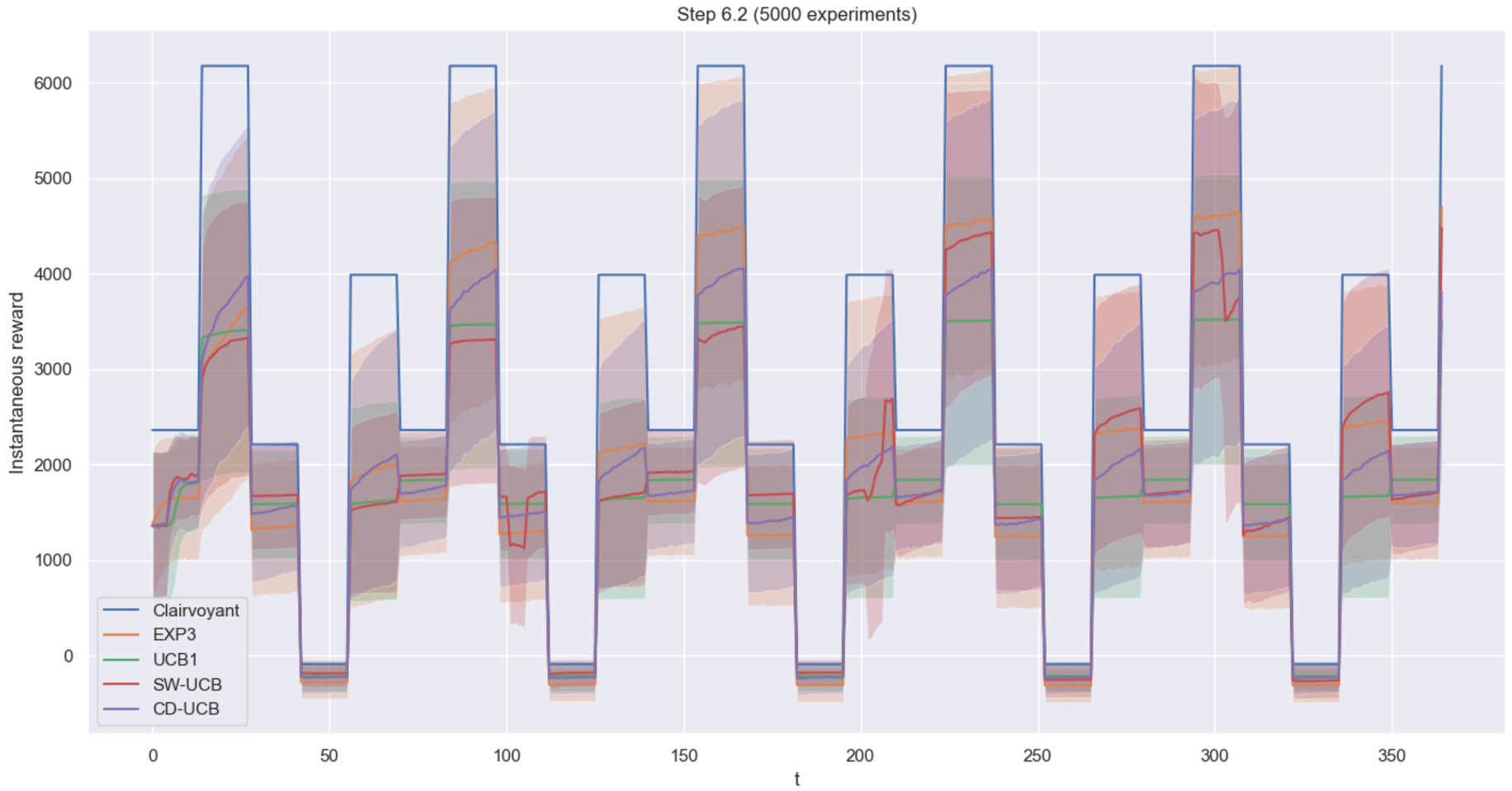
6.1 - Cumulative Reward



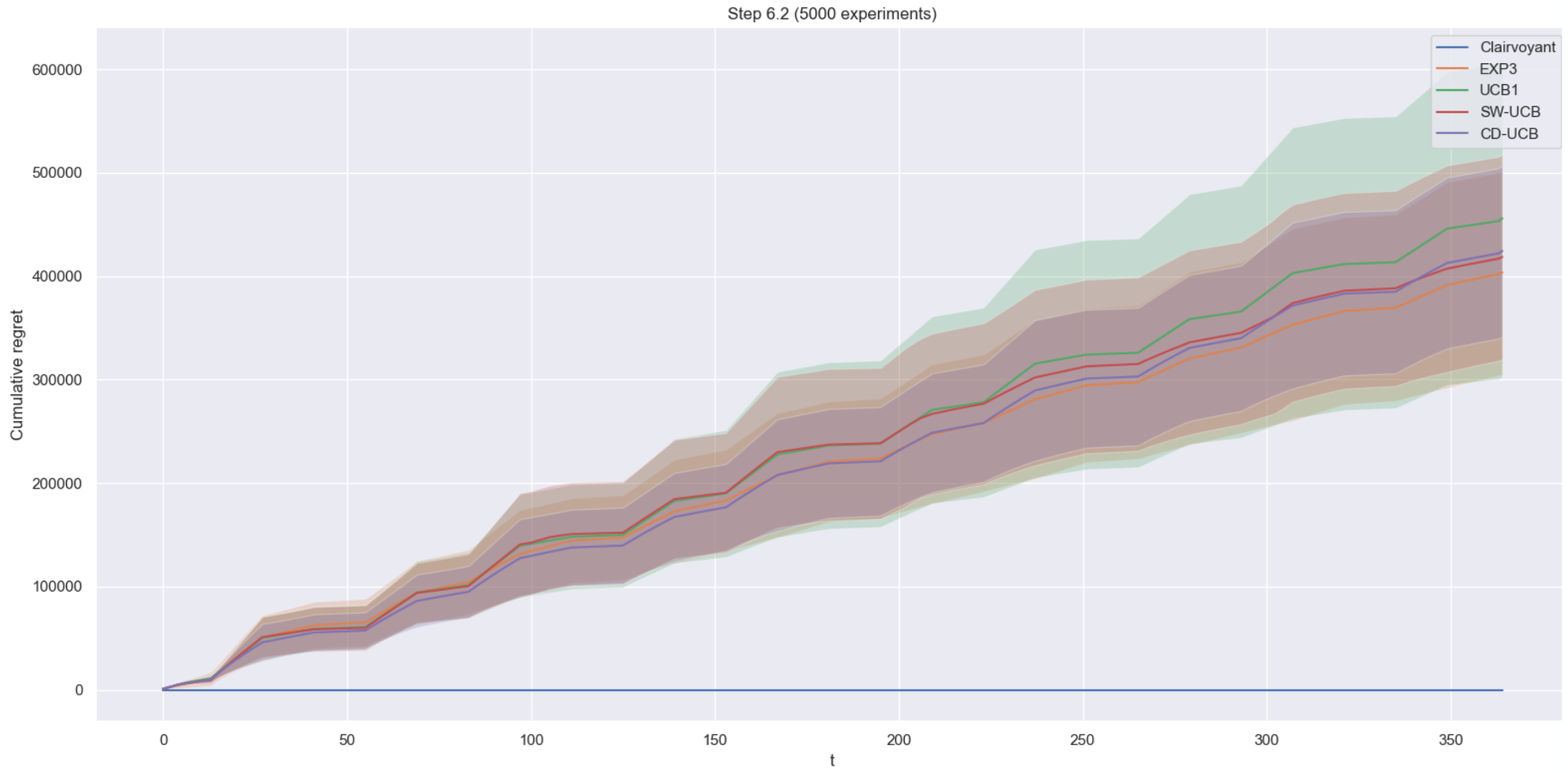
6.2 - Instantaneous Regret



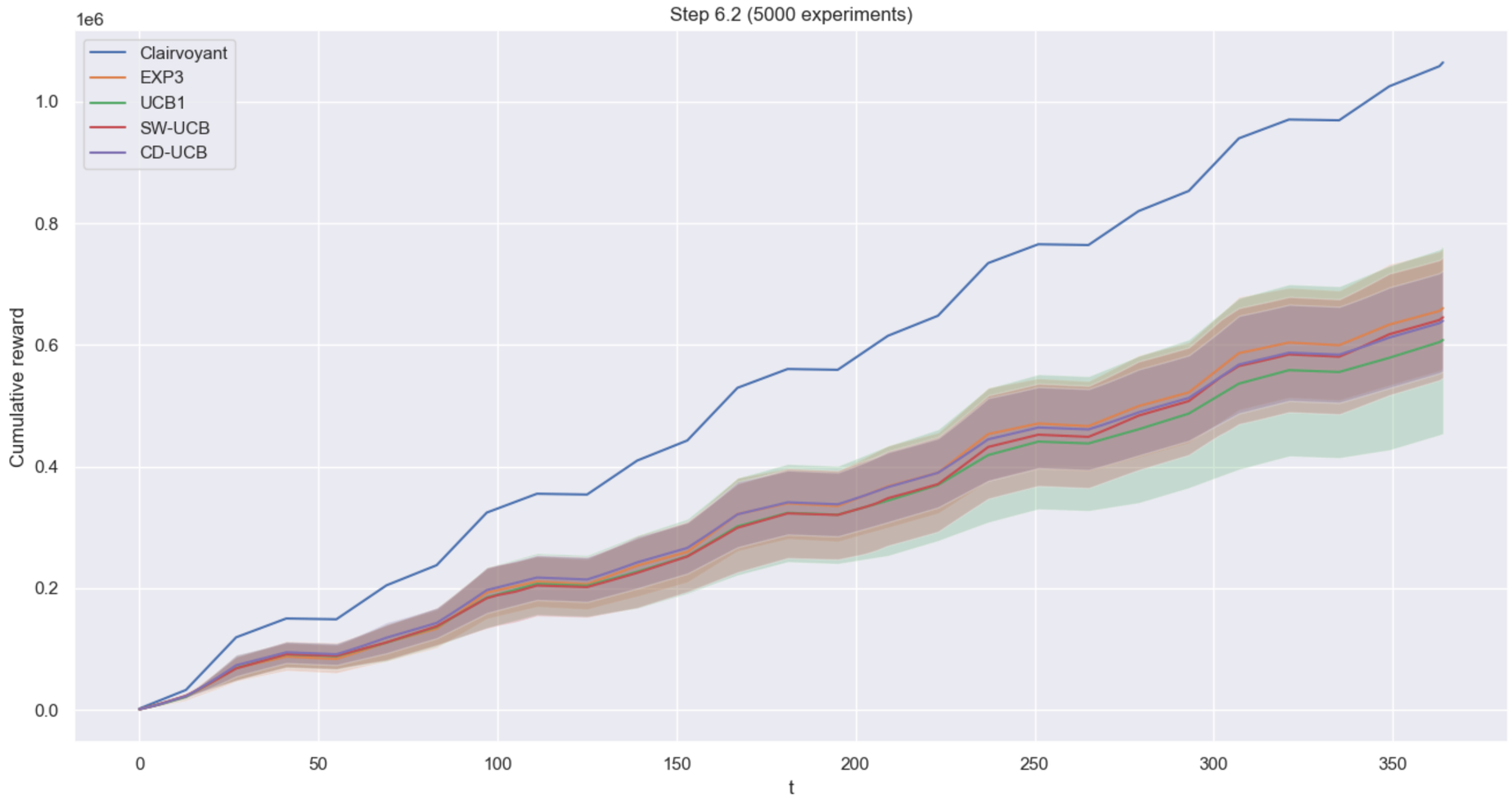
6.2 - Instantaneous Reward



6.2 - Cumulative Regret



6.2 - Cumulative Reward



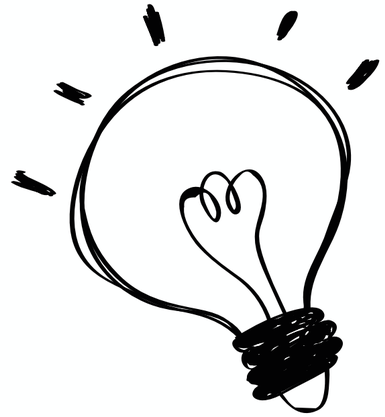
Comments and Conclusions

As we expected in step 6.1, where there is an environment like Step 5, but with the bid fixed, EXP3 performs **worse** than the two non-stationary UCB1.

In the second case, with **5 phases** that change with **high** frequency, EXP3 performed **better** than the non-stationary UCB1.

In both steps, we chose a very small γ for the EXP3 algorithm to **avoid** having too **much exploration and less exploitation.**

Summary
6 Steps



01

Both algorithms have a **logarithmic** cumulative regret

02

Both algorithms have a **logarithmic** cumulative regret

03

The **cascade** application of the first two steps obtains **good** results

04

The case where the **context** is **known** is the **best**. The **context generation** obtains an **excellent** result.

05

SW and **especially CD** manage to make UCB **competitive** even in environments with **abrupt changes**

06

EXP3 outperforms every other algorithm with **very frequent scenario changes**

Improvements

01

Experimenting with **multiple** classes in every step

02

Experimenting with **unbalanced** classes

03

Experimenting with **adversial** environments

GOT QUESTIONS?

Reach out.

Project Github

POLITECNICO
MILANO 1863