Online Learning Application (056894)

Matching - Social Influence

Group 8:
- Andrea Borromini 10665331
- Jacopo Grassi 10869183

04/08/2023

Table of Contents

0
Problem setting

Problem setting and environment (1/4)

Scenario

A company wants to increase the visibility of its products by:

- using social influence techniques to stimulate as many purchases as possible,
- adequately match the reached customers with its different products.

Social environment

Each round corresponds to a day. The time horizon is 365 days.

The company focuses on a network of 30 customers. Some customers might present connections among each other. Each connection is associated to a different activation probability. The activation of the edge determines the social influence propagation.

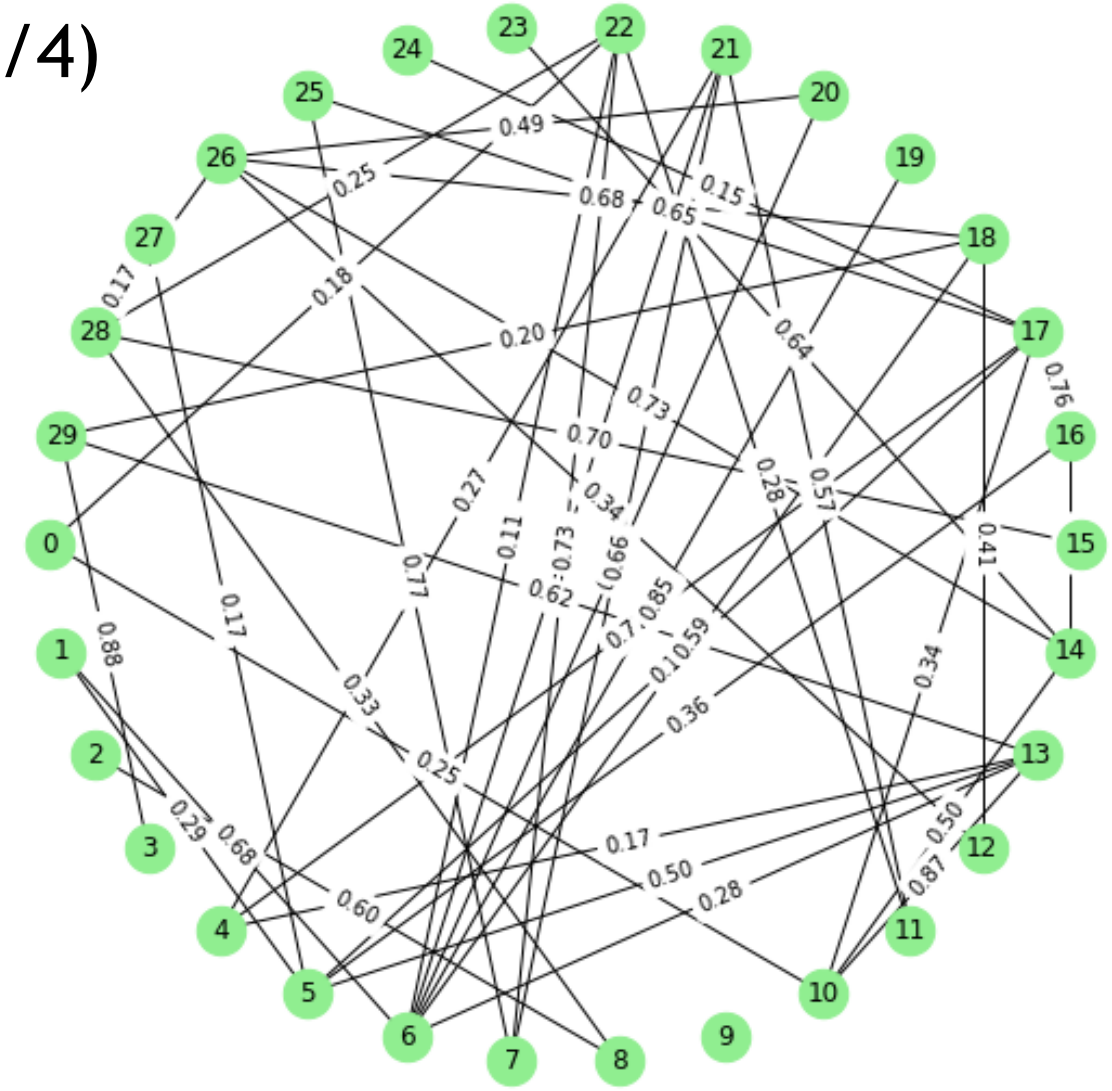
The company can choose three seeds to activate in the social network. The influence phenomenon will propagate starting from them.

Problem setting and environment (2/4)

Edges and their activation probabilities are implemented using a 30 * 30 table. A single customer is associated to a specific row and to a specific column (i.e., customer 0, customer 1, etc.).

	0	1	2	3	…
0	0	P	0	0	…
1	0	0	0	0	…
2	0	0	0	P	…
3	P	0	0	0	…
…	…	…	…	…	…

For instance, this is the probability associated with the activation of the edge between customer 0 and customer 1.



Example of influence network

Problem setting and environment (3/4)

Matching environment

Each customer has two binary features (F1, F2) determining the assignment of the customer to one of three classes (A, B, C). The company sells three different types of products D1, D2, D3.

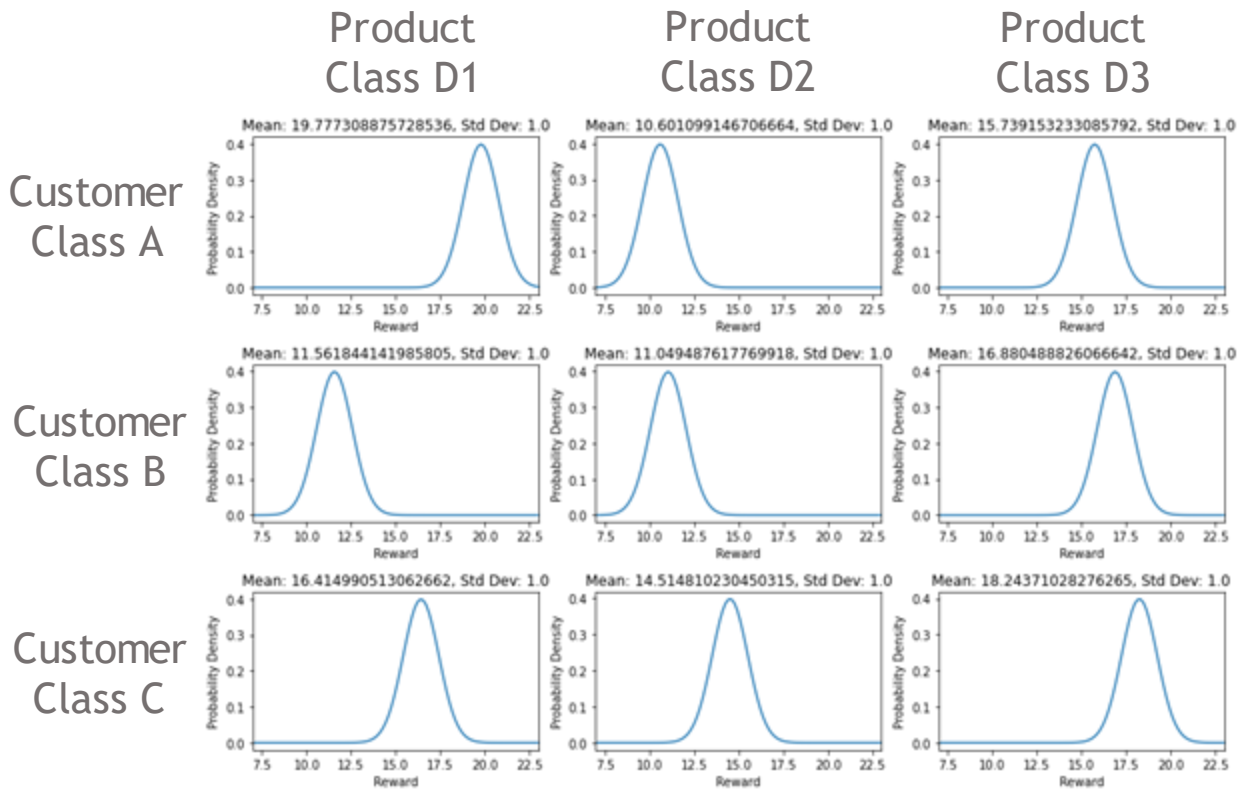
Matching a certain type of product with a certain class of customer leads to a certain reward. For each combination of product class and customer class, the matching reward is modeled as a Gaussian distribution.

At each round, the company has in stock 3 units for each type of product. Each unit of product can be matched only with one customer. After the matching, the customer abandons the experiment, and the stock levels are updated. The matching continues as long as (a) new customers are influenced and make another purchase, and (b) products are not out of stock.

Problem setting and environment (4/4)

For each combination of product type and customer class, the matching reward is modeled as a Gaussian distribution.

When an influenced customer is matched with a certain product, the reward is drawn from the corresponding matching reward distribution.



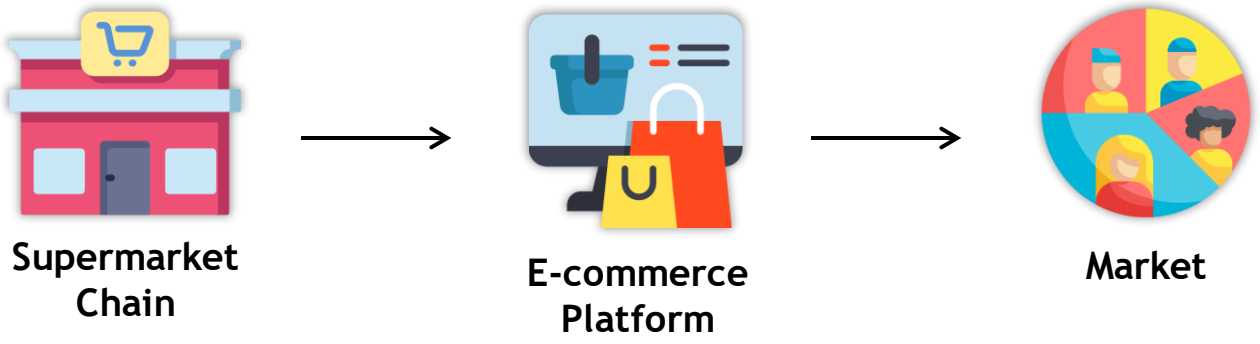
Example of matching reward distributions

1
Motivations and environment design

Realistic Scenario (1/1)

A supermarket chain wants to leverage an e-commerce platform to increase the visibility of its products and the satisfaction/retention of its customers. The company managers come up with the following strategies:

- **Exploiting social influence techniques** to stimulate as many purchases as possible,
- **Matching the reached customers with some appropriate gadget gifts** depending on their features.

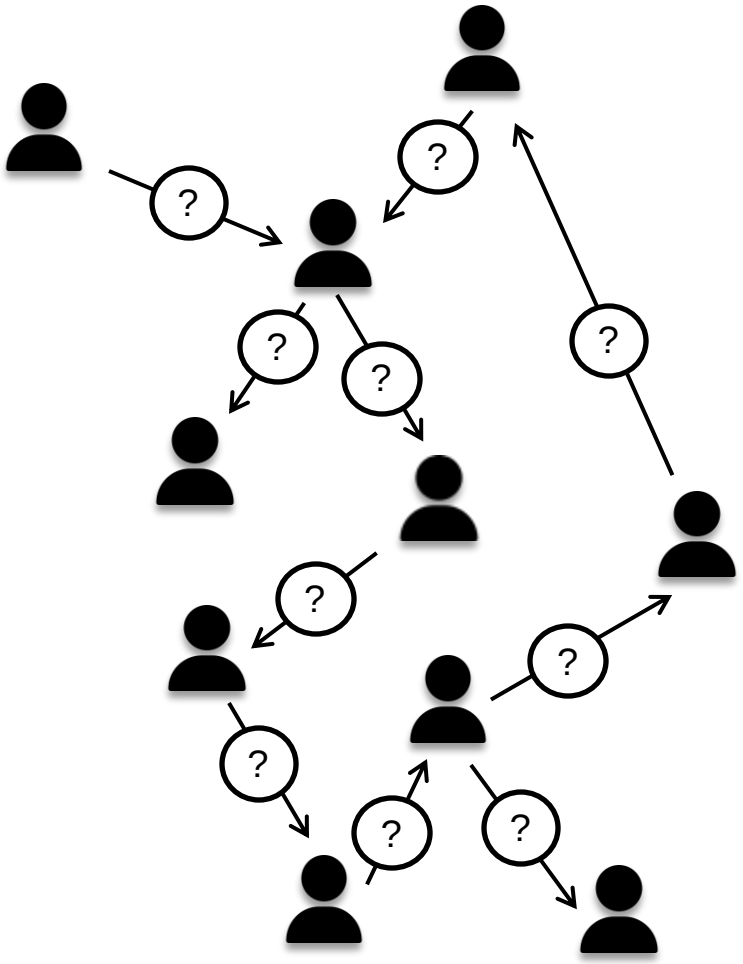


Exploiting Social Influence Techniques (1/1)

Potential customers are part of a social network in which – with a certain probability – the current purchase of one customer can stimulate the future purchase of another.

By observing the probabilities through which potential customers influence each other (i.e. activation probabilities), the company can identify the most relevant customers to target (i.e. optimal seeds) in order to generate a fruitful "cascade" of purchases.

Unfortunately, such activation probabilities may be **unknown… But we can try to estimate them using Bandit Algorithms.**



Customer-gadget matching (1/2)

Every day, once the optimal customers are activated, the social influence propagation will yield a queue of subsequently activated clients. To boost their satisfaction and increase the visibility of the provided services, the company decides to offer them some free **gadget gifts**.
Gadget gifts are divided into the **three categories** shown below**.** The company has decided to gift only 3 items (for each category) per day.

Electronics & Tech

Craft and DIY

Fashion & Lifestyle

Customer-gadget matching (2/2)

Moreover, the company is assumed to distinguish customers depending on the following binary features:
1. **Age-range**: Youngsters (<= 25 years old), Adults (> 25 years old)
2. **Gender** (Male or Female)

As a result of the combination of the two features, three classes of customers are outlined:

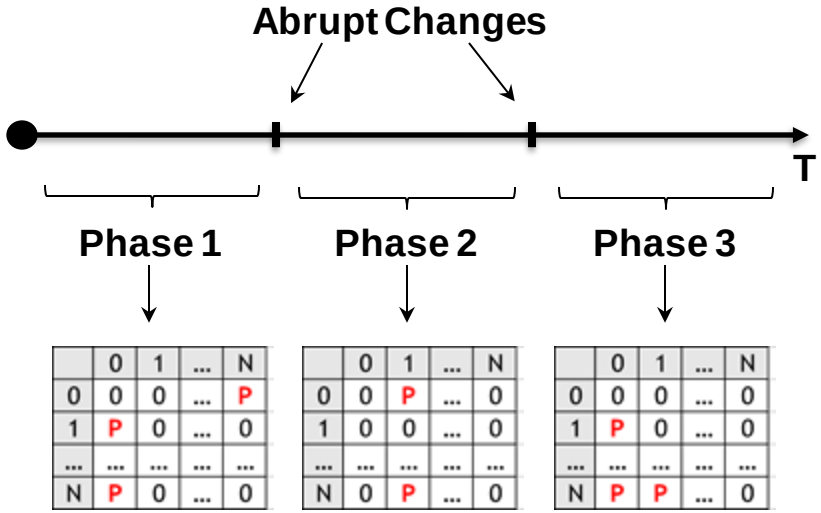
A. **Youngsters** (generally more interested in **Electronics and Tech gadgets**)
B. **Male Adults** (more likely to appreciate **Craft and DIY gadgets**)
C. **Female Adults** (particularly into **Fashion and Lifestyle gadgets**)

These dynamics of compatibility between gadget type and category of customer is paramount for the company. In fact, the better the matching between gadget type and customer, the higher the probability of the customer purchasing again and/or influencing its network (of friends, relatives or colleagues) to also purchase through the platform. In other words, the better the matching, the higher the company's reward.
Unfortunately, the distributions describing the adequacy (i.e. reward) of the customer-gadget matching are **unknown… But we can use Matching Bandit Algorithms to estimate them**.

Non-stationary social influence (1/1)

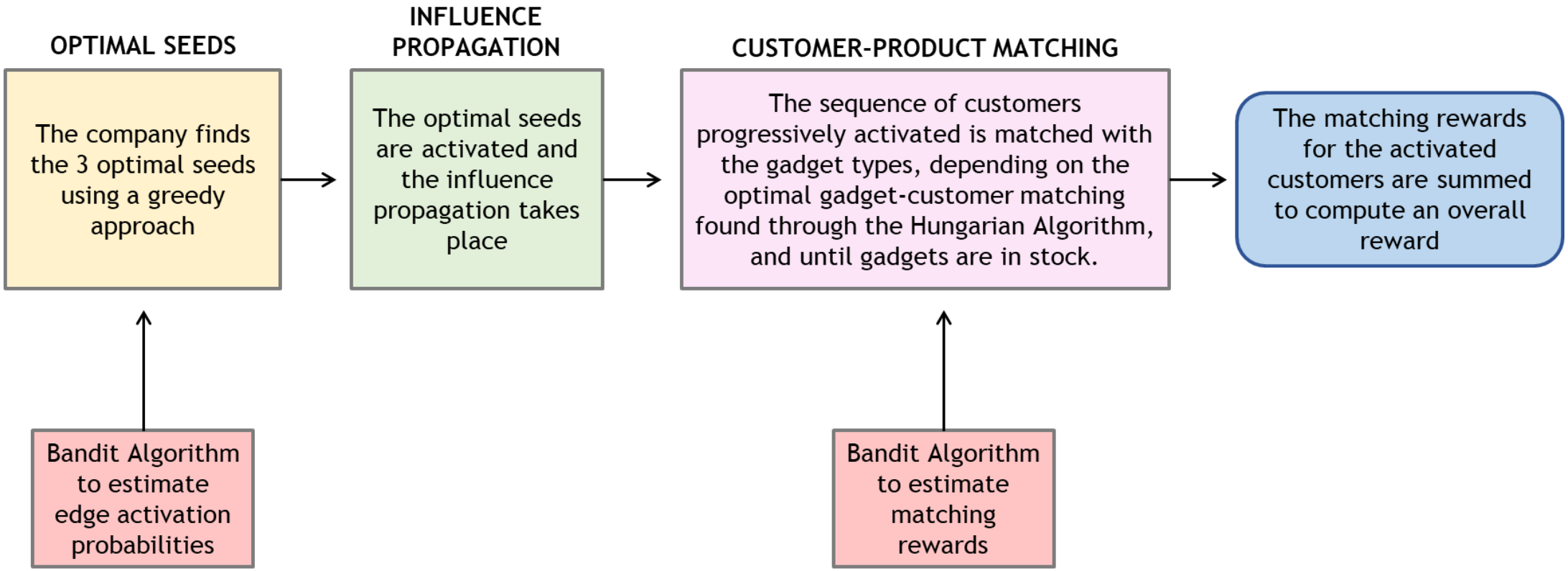
Finally, the company is aware of the fact that, through social interactions, customers are able to influence their friends, relatives and colleagues in different ways during distinct periods of the year. For instance, we assume that the company expects the year to be divided into three phases, by two abrupt changes:

- **January - April** (in which, for example, youngsters are more likely to influence youngsters and adults are more likely to influence adults due to their permanence in schools, universities or offices).
- **May - August** (in which the summer encourages youngster-adult socializations increasing their influence on each other)
- **September - December** (in which the Christmas holiday period generally boosts influence among all categories of customers)



Overall process (1/1)

Every day of the time horizon, the following process takes place:



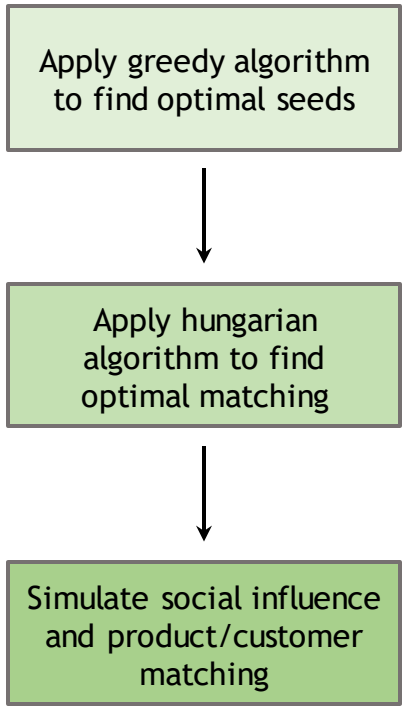
2
Clairvoyant optimization algorithm

Clairvoyant optimization algorithm (1/9)

In our problem, the clairvoyant algorithms aims to:

1. Find the optimal seeds through a greedy algorithm starting from an input (**true or estimated**) edge activation probability table. Optimal seeds are assumed to be seeds that activate the largest number of nodes in the overall influence propagation.
2. Find the optimal matching between customer classes and product classes using the Hungarian algorithm on some input (**true or estimated**) expected matching rewards.
3. Simulate multiple runs of the social influence process (starting from the fixed initial seeds) using the **true** edge activation probabilities. In each run, given 3 items for each product class:
   - for each set of activated nodes, extract a reward from the **true** distributions associated to the appropriate optimal matching of the corresponding customer class and product class. If the optimal matching is unfeasible due to out-of-stock conditions, choose a random in-stock product to assign. Update the items availability.
   - Repeat the process until no new customer is influenced or until no items are left in stock.

   Average the runs' rewards to obtain an expectation.



Clairvoyant optimization algorithm (2/9)

simulate_episode – function that

- receives in **input** (1) a graph probability matrix, (2) a list of seeds and a (3) maximum number of steps,
- and **outputs** the history of the activations of nodes starting from the seeds and the list of the activated nodes at the end of the propagation.

List of seeds (e.g. [1, 13, 22]

Max. number of steps for the propagation.

	0	1	2	3	…
0	0	P	0	0	…
1	0	0	0	0	…
2	0	0	0	P	…
3	P	0	0	0	…
…	…	…	…	…	…

```
def simulate_episode(init_prob_matrix, seeds:list, max_steps):
    ...
    return history
```

Each row in the history array corresponds to a stage of the propagation and shows the new activated nodes in that steps.

→

```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.,
        0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

Clairvoyant optimization algorithm (3/9)

PSEUDOCODE simulate_episode

- Create a copy of the graph probability matrix
- Initialize the first row of the "history" array displaying a “1” in the position of the activated seeds and “0” otherwise.
- While the number of steps is less than the maximum number of steps in input and as long as the current step activates some new nodes:
  - For the activated nodes in the current step, extract a sample from a uniform distribution between 0 and 1. If the value is smaller than the edge probability associated to the node in the copy of the graph probability matrix, the node is saved as a newly activated node.
  - In the copy of the probability matrix, zero the row corresponding to the newly activated nodes.
  - Add the newly activated node to the "history" rows.
  - Repeat the process on the newly activated nodes in a new step.
- Return “history” array and active nodes at the end of the propagation.

Clairvoyant optimization algorithm (4/9)

test_seed – function that

- receives in **input** (1) some seeds, (2) a graph probability matrix, (3) a number of trials k, (4) the maximum number of steps,
- and **outputs** the average reward over the k trials (where the reward is the number of activated nodes in the propagation starting from the given seeds).

PSEUDOCODE test_seed

- Through the **simulate_episode** function, simulate the influence propagation process starting from the given seeds k times and sum their rewards (number of overall activated nodes in the propagation).
- Return the average reward for the seeds.

```
test_seed([1,13,22], graph_probabilities, 100, 10)
```

```
11.61
```

Example of test_seed application

Clairvoyant optimization algorithm (5/9)

greedy_algorithm – function that:

- receives in **input** the (1) graph probability matrix, (2) a budget (number of seeds to find), (3) a number of trials k, (4) the maximum number of steps for the influence propagation,
- and **outputs** the three optimal seeds. (nodes that if activated maximize the number of consequently activated nodes).

```
greedy_algorithm(graph_probabilities, 3, 100, 10)
```

```
[23, 20, 9]
```

Example of test_seed application

PSEUDOCODE greedy_algorithm

- Initialize an empty list of seeds.
- For each unit allowed by the budget:
  - Use **test_seed** function to compute the average reward (over k trials) given by considering seed each single node of the graph probability matrix.
  - Choose the node that maximizes the reward as optimal seed (add it to the list of seeds)
- Return list of seeds.

Clairvoyant optimization algorithm (6/9)

hungarian_algorithm – function that:

- receives in **input** a matrix showing the rewards of matching elements of a bipartite graph (i.e. classes of product with classes of customers),
- and **outputs** a matrix of equal dimensions where 1 represents an assignment, 0 otherwise.

```
[[17.21409833 19.01568037 17.05831714]
 [18.50147206 14.47550161 19.32457588]
 [13.71069826 15.45953857 10.42237602]]
```

```
hungarian_algorithm(means)[1]
```

```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

Example of hungarian_algorithm application

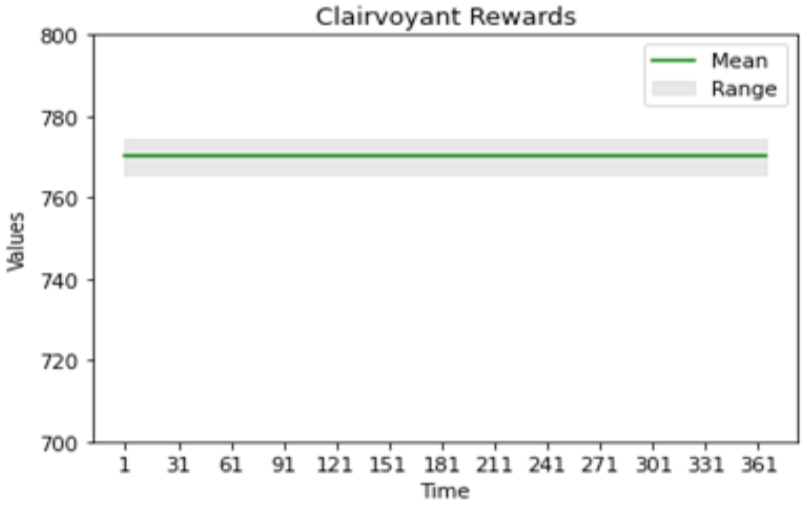
PSEUDOCODE hungarian_algorithm

- Create a copy of the matching reward matrix to avoid modifying the original one.
- Use *linear_sum_assignment* function (scipy.optimize module) to solve the assignment problem.
- Return a matrix having same dimensions of the original one, where 1 indicates a match, 0 a non-match.

Clairvoyant optimization algorithm (7/9)

clairvoyant – function that:

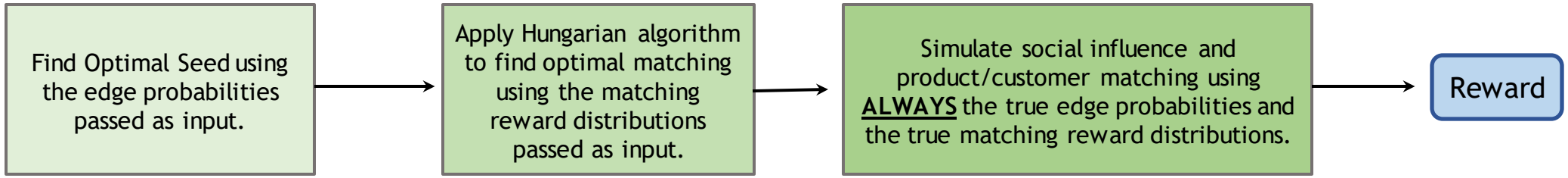
- receives in **input** (1) a given graph probability matrix, (2) the true graph probability matrix, (3) the assignment of customers to classes, (4) a given set of expected matching rewards (mean and std_dev), (5) the true expected matching rewards, (6) a number of experiments.
- and **outputs** mean and standard deviation of the reward generated by (a) finding a set of optimal seeds using the greedy algorithm, (b) simulating influence propagation with our episode simulator, and (c) finding the best matching using the Hungarian algorithm. To make the output closer to the real optimum, we run our clairvoyant function multiple times and choose the highest reward mean.



Example of clairvoyant rewards in time

Clairvoyant optimization algorithm (8/9)

Notice how the decision-making processes for determining the initial seeds and the best customer-product matching are performed by considering the **given** edge activation probability tables and the **given** expected matching rewards, while the corresponding overall reward is computed taking in consideration the **true** activation probabilities and **true** matching rewards.



By passing as input the true edge probability tables and the true matching reward distributions, we make the algorithm fully Clairvoyant, hence obtaining the overall Optimum. Conversely, if we pass as input the estimated edge probabilities and estimated matching rewards, we obtain the reward we would collect in a situation of uncertainty.

Clairvoyant optimization algorithm (9/9)

PSEUDOCODE clairvoyant

1. Find three optimum seeds using **greedy_algorithm** starting on the **given** activation probabilities.
2. Starting from such seeds, use the **true** activation probabilities to simulate the influence process with **simulate_episode** and build a list showing the nodes that activate (in order of activation).
3. Use the **given** matching rewards to solve the matching problem using the **hungarian_algorithm** and find the "expected" optimal matching.
4. As long as there are units in stock for the product classes, for each activated customer:
    - If the optimal product class for the customer is not out of stock, extract a reward from the corresponding **true** matching reward distribution. Update the number of items available for that product class.
    - If the optimal product class for the customer is out of stock, assign another class of product at random and extract a reward from the corresponding **true** matching reward distribution. Update the number of items available for that product class.
    - Sum up the rewards to obtain the overall reward of the propagation + matching.
5. Potentially repeat the points 2, 3, 4 many times (in many experiments), and return the mean and standard deviation of the rewards obtained.

3
Learning for social influence

Upper Confidence Bound UCB Learner (1/3)

PSEUDOCODE

1. Initialize a Bernoulli distribution for each arm (edge), with mean equal to the corresponding activation probability in the initial graph probability table.
2. Play every arm once. When an arm is played, we extract a sample from the corresponding Bernoulli distribution. In the code, this is implemented through the Environment class.
3. Update the mean and the upper confidence bound of the played arms.
4. At every subsequent round t, play the arm displaying the maximum confidence bound. If multiple arms share the maximum value of the bound, choose and play one of them at random. Once an arm has been played, store the reward.

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

5. Repeat steps 2 and 3 until the end of the time horizon.

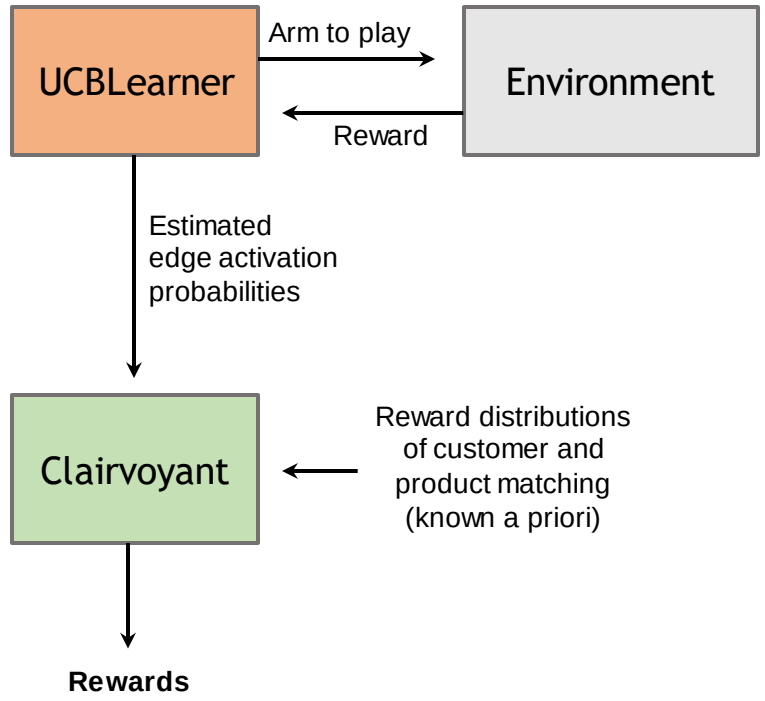
Upper Confidence Bound UCB Learner (2/3)

In our scenario, each arm represents a possible edge between two customers, while the Bernoulli outcome indicates the activation or non-activation of such edge.

At each round:

For each arm, the algorithm will keep track of the empirical means* of the Bernoulli samples.

- **Such means are used as estimates of the edge activation probability** and combined in a 30*30 table.
- For each round, the current table is passed as input to the Clairvoyant function, which (a) finds the optimal seeds, (b) simulates the influence propagation, and (c) determines the reward obtained by the optimal assignment of customers' classes and products' types. The reward distributions for matching products and customers are assumed to be known.



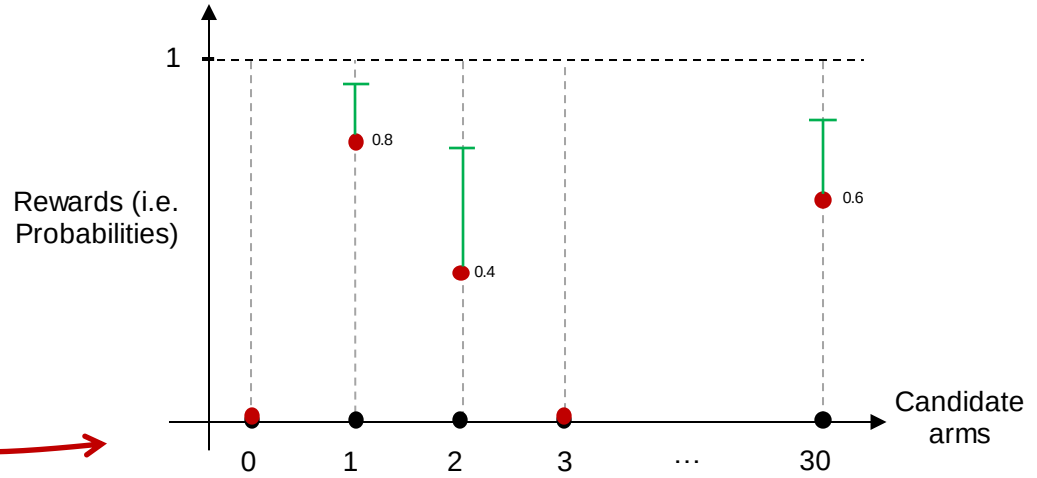
*We purposefully chose to avoid using the Upper Confidence Bounds to prevent the distortion of the influence propagation process caused by the excessively high edge activation probability estimates (i.e., outside range [0,1]).

Upper Confidence Bound UCB Learner (3/3)

Considering that the total number of possible edge probabilities to assess is 900 (30*30) and that many of them are equal to 0, to avoid sparsity we initialized a single Learner for each row in the graph probability table, reducing the number of arms to 30.

The same has been done for all learners in this project.

	0	1	2	3	…	30
0	0	**0.8**	**0.4**	0	…	**0.6**
1	0	0	0	0.1	…	0
…	…	…	…	…	…	…
30	0	0	0	0	…	0



Thompson Sampling TS Learner (1/2)

PSEUDOCODE

1. Initialize a Bernoulli distribution for each arm, with mean equal to the corresponding activation probability in the initial graph probability table.

2. Initialize a Beta distribution for each arm, with parameters α alpha = 1 and ß beta = 1 (i.e., the Beta distribution at round 0 is a uniform distribution).

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

3. At every round, for each arm, extract a sample from the corresponding beta distribution.

4. Play the arm with the highest sample (i.e., extract a reward from the corresponding Bernoulli distribution). In the code, this is implemented through the Environment class.

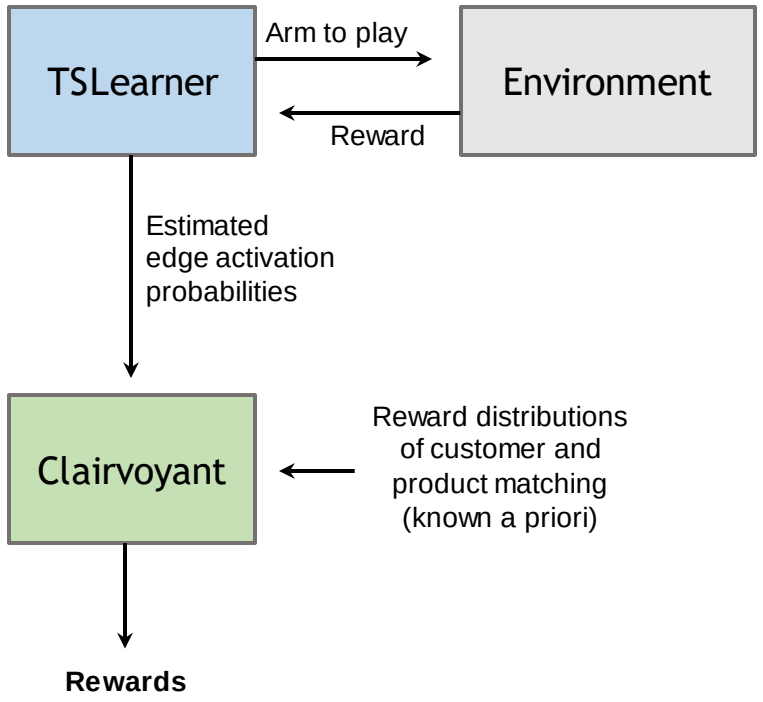
5. Update the corresponding Beta distribution such that:
   - The alpha parameter must reflect the number of Bernoulli successes (1)
   - The beta parameter must reflect the number of Bernoulli unsuccesses (0)

Thompson Sampling TS Learner (2/2)

Like before, each arm represents a possible edge between two customers, while the Bernoulli outcome indicates the activation or non-activation of such edge.

At each round:

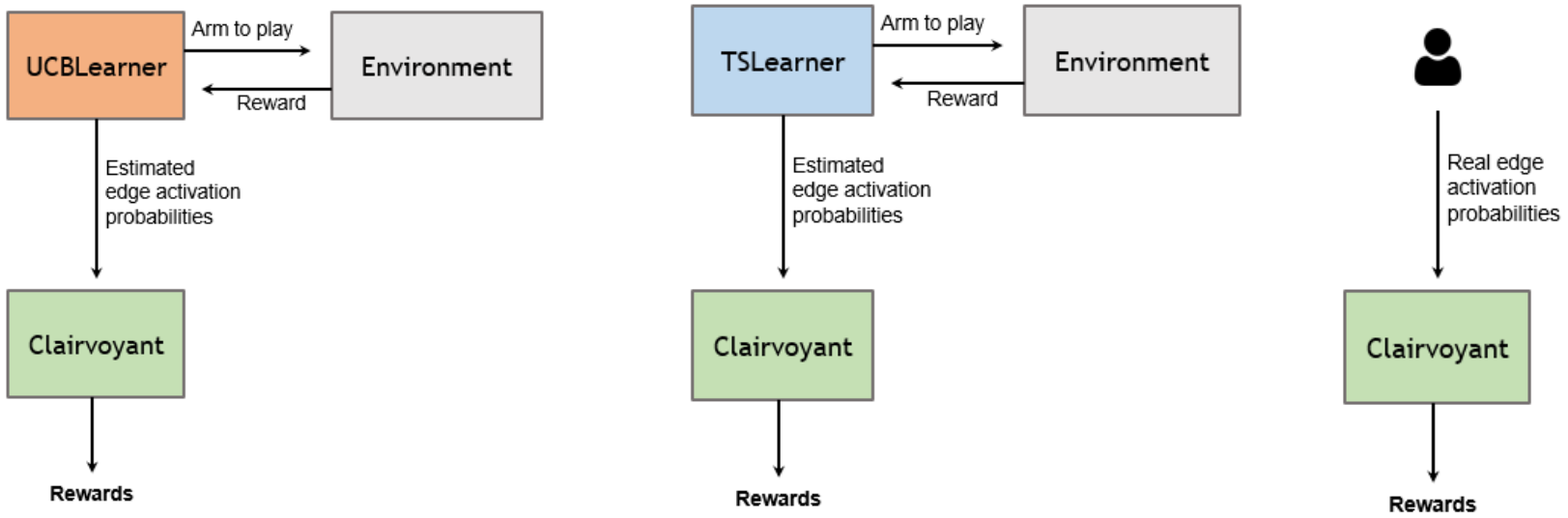
- **For each arm, the algorithm will keep track of the empirical mean of the Beta distribution** (Mean = α / (α + β)).
- **Such means are used as estimates of the edge activation probability** and combined in a 30*30 table.
- Exactly as before, for each round, the current table is passed as input to the Clairvoyant function, which (a) finds the optimal seeds, (b) simulates the influence propagation, and (c) determines the reward obtained by the optimal assignment of customers' classes and products' types. The reward distributions for assigning products and customers are assumed to be known.



Comparing UCB, TS and Clairvoyant (1/8)

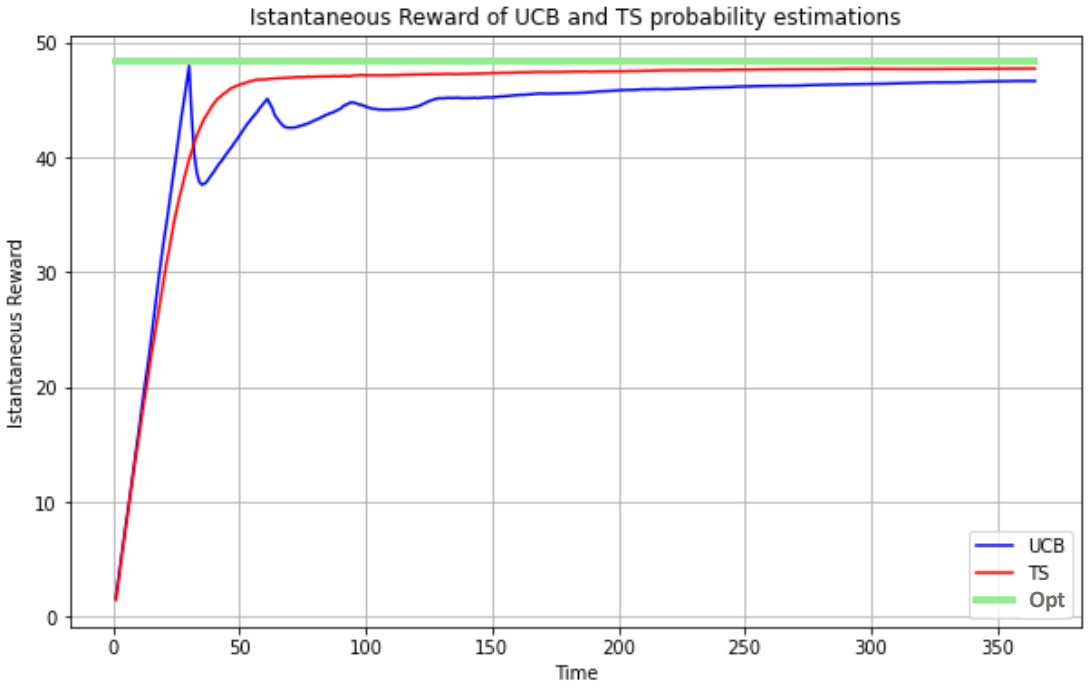
At each round, we computed and compared:

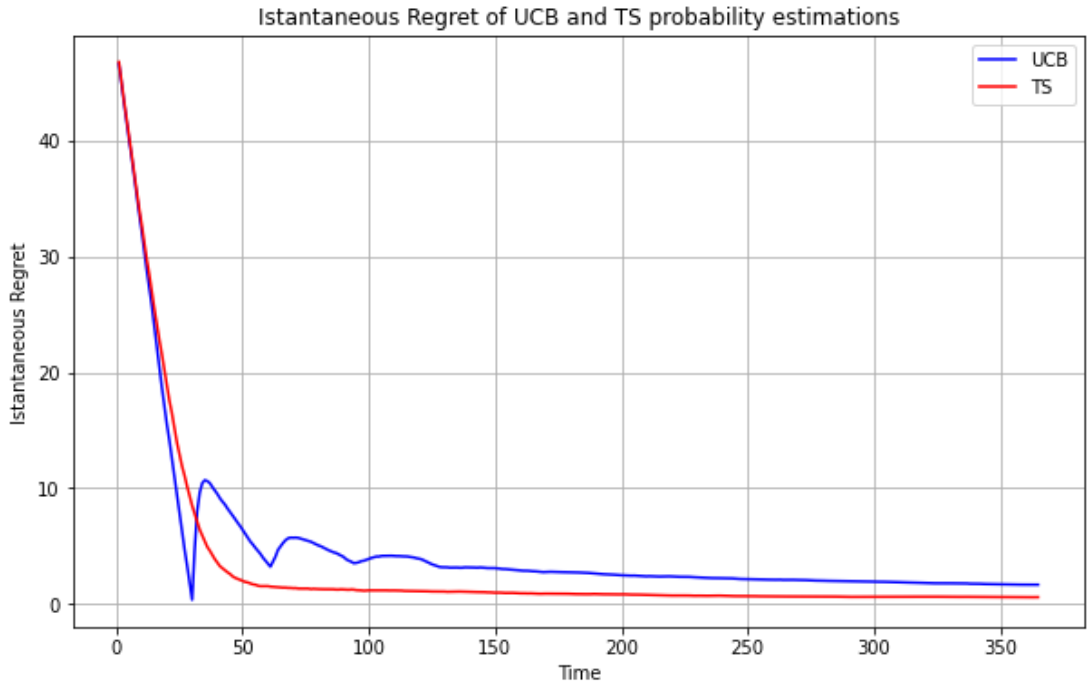
1. The rewards obtained by passing as input the activation probabilities (estimated with UCB) to the Clairvoyant algorithm.
2. The rewards obtained by passing as input the activation probabilities (estimated with TS) to the Clairvoyant algorithm.
3. The rewards obtained by feeding the Clairvoyant algorithm with the real activation probabilities.



Comparing UCB, TS and Clairvoyant (2/8)

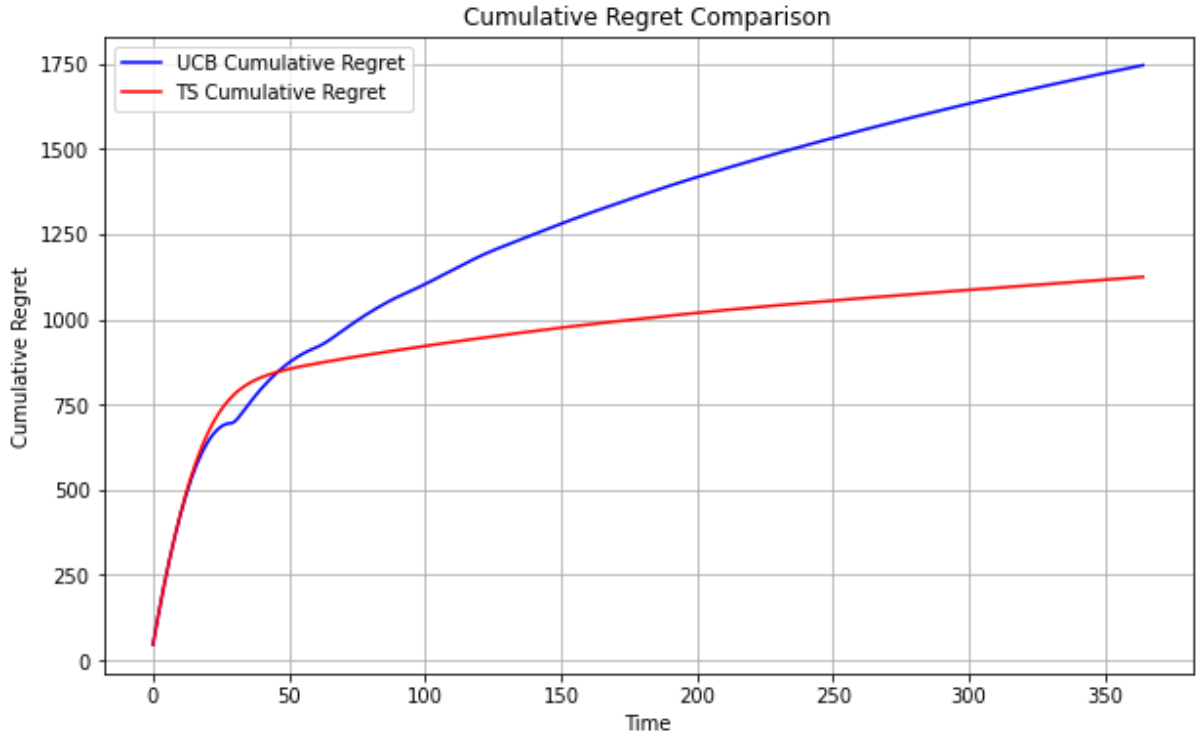
The following is a comparison of UCB and TS instantaneous rewards and regrets with respect to the probability estimation task. In each round, the reward is assessed by computing the sum of the edge activation probabilities in our network. The (green) optimum corresponds to the sum of the probabilities in the true graph probability table.





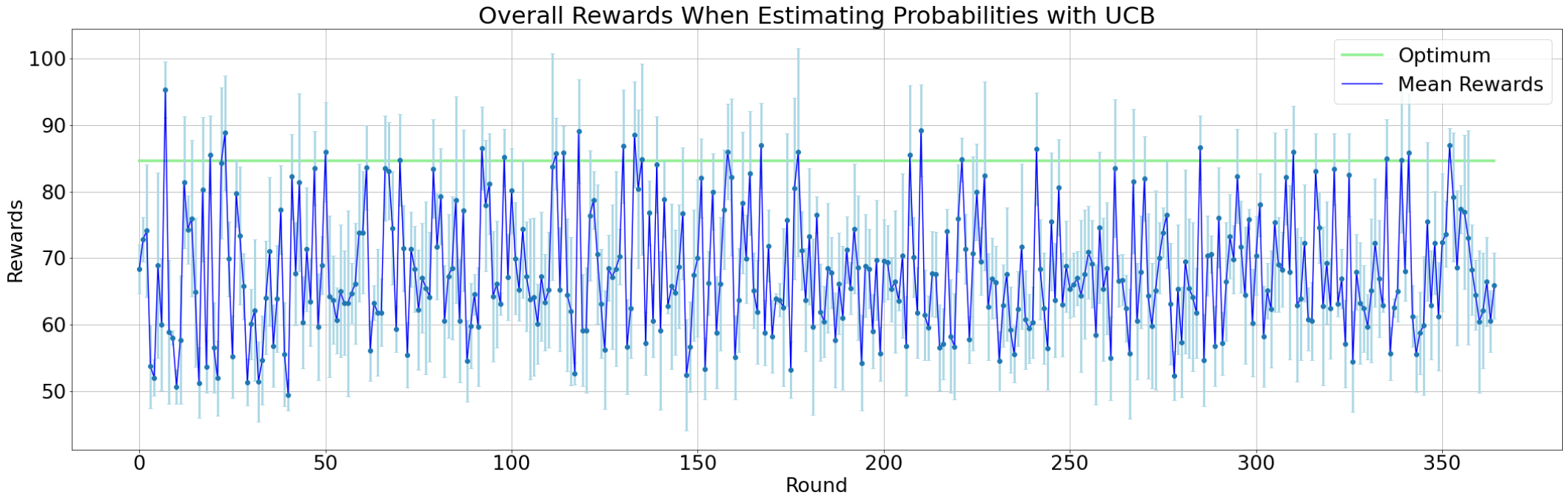
Comparing UCB, TS and Clairvoyant (3/8)

By comparing the cumulative regrets of UCB and TS in the task of edge probability estimation, TS appears to outperform UCB. Both regrets are increasing sublinearly in time.



Comparing UCB, TS and Clairvoyant (4/8)

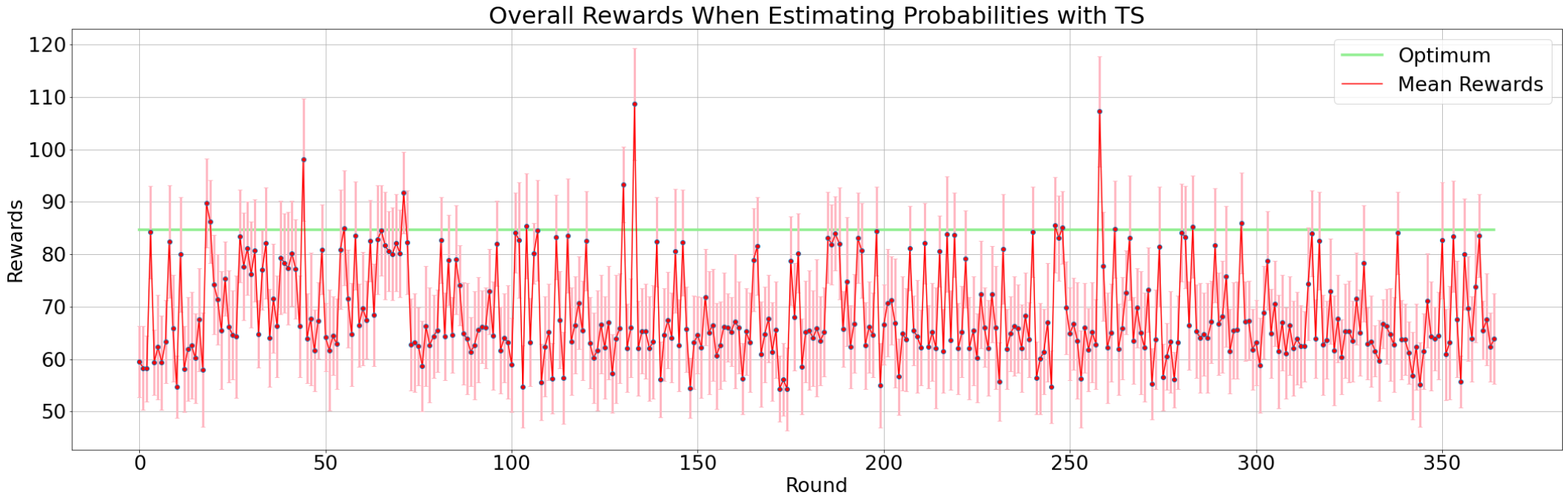
Overall rewards (influence simulation + matching) when estimating edge activation probabilities with **UCB**



To find the **Optimum**, we simulated the influence propagation and matching task multiple times, using the true edge activation probabilities and the true expected matching rewards. Finally, we selected the maximum reward obtained from all iterations.

Comparing UCB, TS and Clairvoyant (5/8)

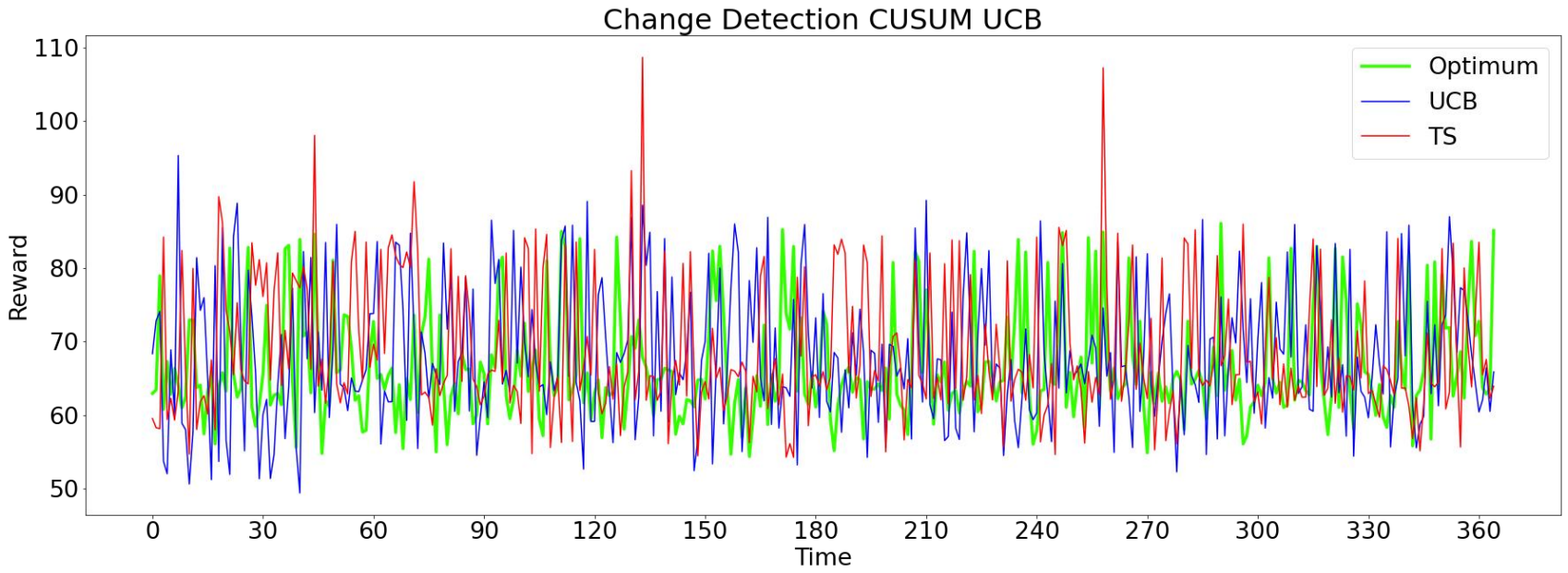
Overall rewards (influence simulation + matching) when estimating edge activation probabilities with **TS**



To find the **Optimum**, we simulated the influence propagation and matching task multiple times, using the true edge activation probabilities and the true expected matching rewards. Finally, we selected the maximum reward obtained from all iterations.

Comparing UCB, TS and Clairvoyant (6/8)

Overall rewards when estimating edge activation probabilities with **UCB**, **TS** and when using the **real probabilities**.



In this case, the optimum was computed by finding the optimal seeds and simulating the influence+matching task at each round.

Comparing UCB, TS and Clairvoyant (7/8)

Considering that:

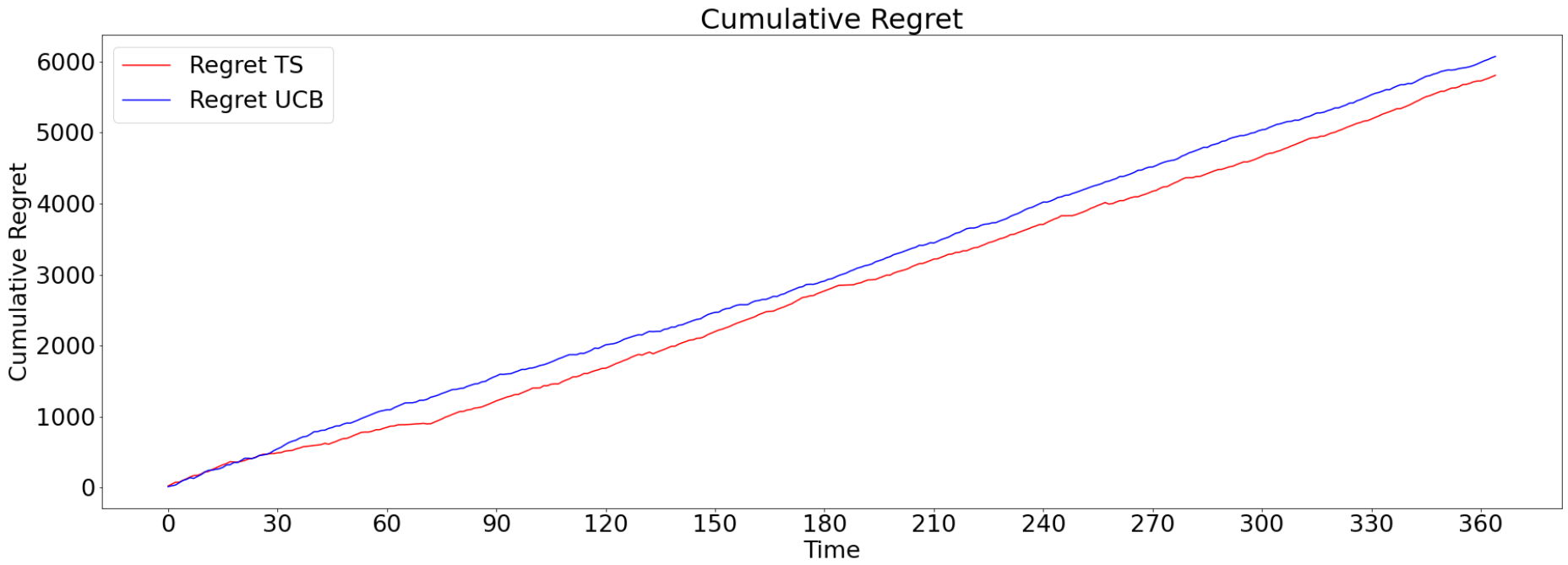
- the instantaneous regrets of UCB and TS in estimating the edge activation probabilities are decreasing,
- the assignment problem was solved through a Hungarian Algorithm…

We expect the volatility of the overall rewards to be caused by:
- the approximative results of our greedy optimum seeds finder,
- the uncertain results of the propagation problem.

This hypothesis is supported by the figure in the previous slide, showcasing the high variability of the overall rewards even when using the true set of edge activation probabilities. Increasing the number of simulations in finding seeds and simulating influence will probably yield more stable results (but also higher computational complexity).

Comparing UCB, TS and Clairvoyant (8/8)

The previous setting has generated the following Cumulative Regret for **TS** and **UCB**. Differently from the cumulative regret computed only for the task of edge probability estimation, the overall problem regret appears to increase linearly in time.



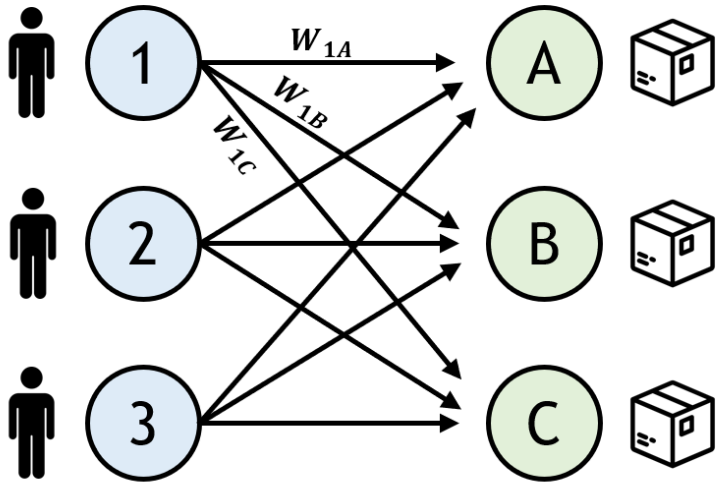
4
Learning for matching

Matching UCB Learner and Matching TS Learner (1/1)

PSEUDOCODE

The proposed matching UCB and TS learners follow the same mechanism of basic UCB and TS algorithms. The difference lies in playing not just a single arm, but a subset of arms (**superarm**). In order to be selected for play, the superarm needs to meet specific combinatorial conditions. The reward of a superarm is the sum of the rewards yielded by the included arms.

In the context of this assignment, the combinatorial problem to solve is a **matching problem**. The 3 customer classes and 3 product types are positioned on the two sides of a **bipartite weighted graph**. Weights represent the means of the reward function associated to linking customer class *i* to product type *j*. In other words, each edge is associated to a random variable whose expected value is unknown and has to be estimated. Such random variables will represent the arms of our UCB and TS matching learners.



Means of Reward Distribution for each assignment

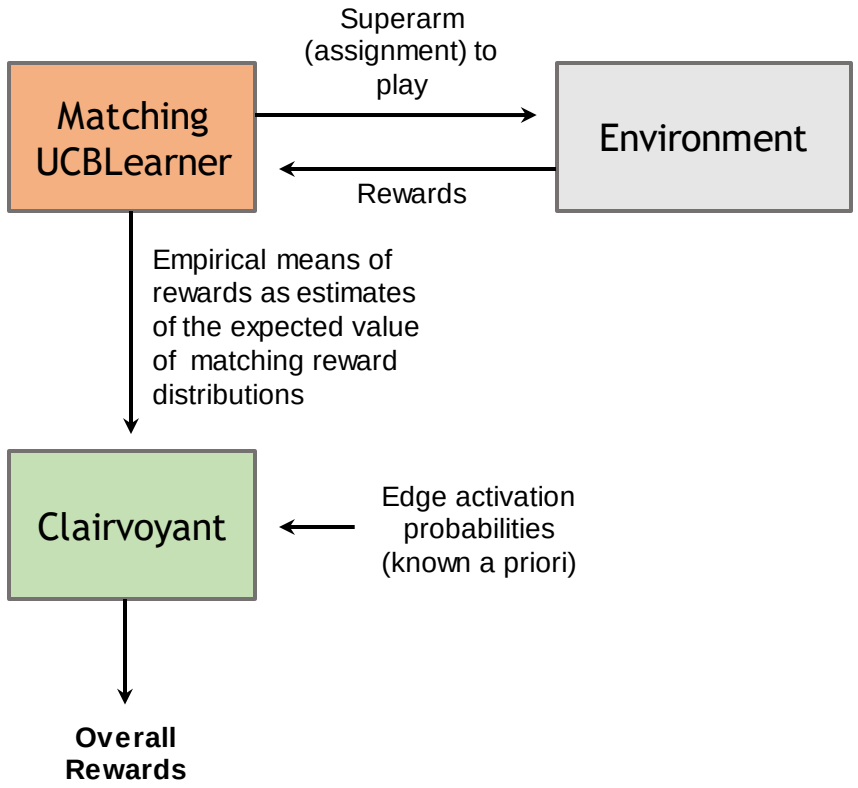
	1	2	3
A	W_{1A}	W_{2A}	W_{3A}
B	W_{1B}	W_{2B}	W_{3B}
C	W_{1C}	W_{2C}	W_{3C}

Matching UCB Learner (1/1)

Arms are edges linking product type *j* with customer class *i* in the bipartite graph. The rewards are Bernoulli samples drawn from the reward distribution associated to linking *j* with *i*.

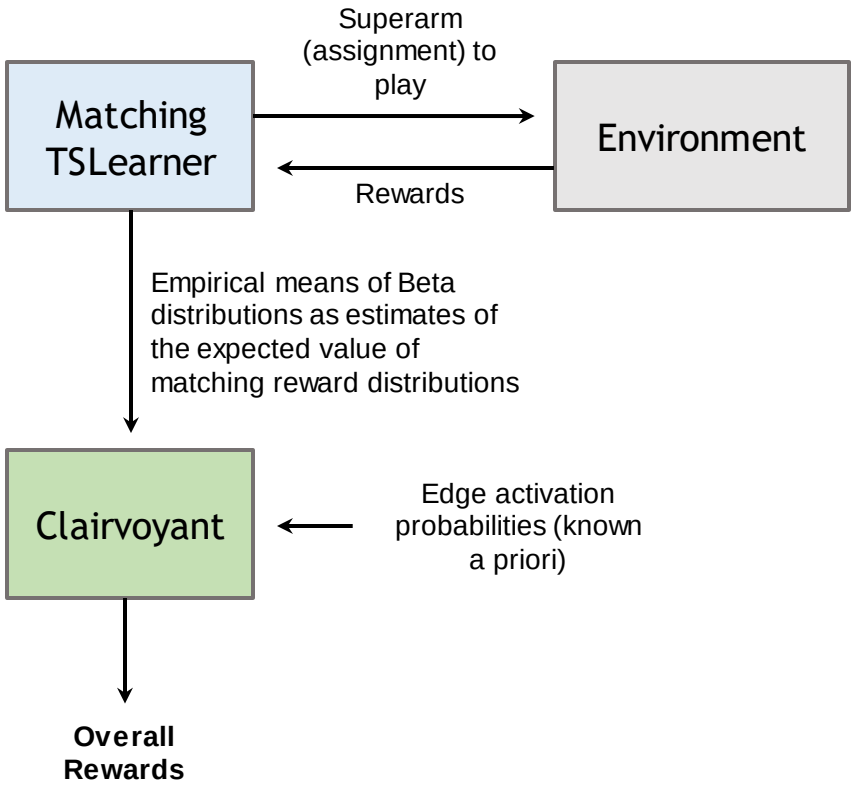
After selecting the feasible superarm to play, at each round:

- For each arm, the algorithm will keep track of the empirical means of the Bernoulli samples.
- Such means are used as estimates of W_{ij}, the expected value of the reward distribution associated to linking *j* with *i*.
- At each round, the means are passed as input to the Clairvoyant function, which:
  - finds the optimal seeds using the true edge activation probabilities (assumed to be known a priori) and simulates the influence propagation,
  - determines the reward obtained by the most adequate product-customer matching (obtained through the estimated matching rewards)



Matching TS Learner (1/1)

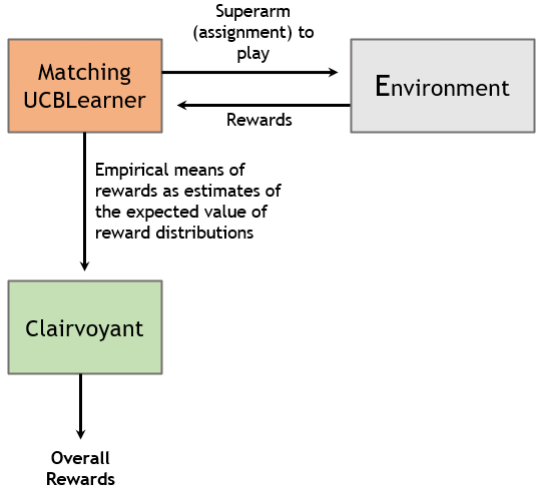
The setting is the same as before, but this time, instead of using the Bernoulli reward empirical means as estimates for the matching expected value, the TSLearner class utilizes its Beta distributions' means.

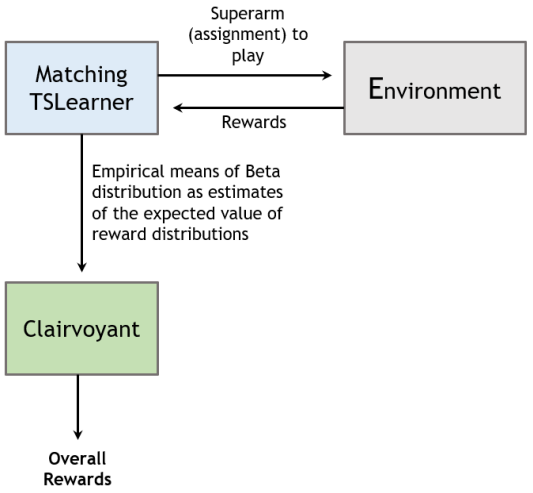


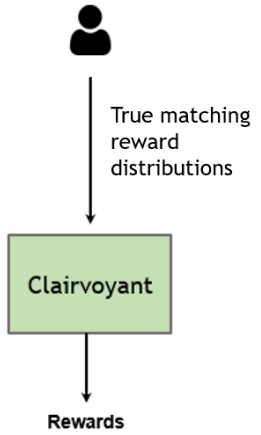
Comparing Matching UCB, TS and Clairvoyant (1/4)

While edges' activation probabilities are assumed to be known a priori… At each round, we computed and compared:

1. The rewards obtained by passing as input the expected value of the matching rewards (estimated with UCB) to the Clairvoyant algorithm.
2. The rewards obtained by passing as input the expected value of the matching rewards (estimated with TS) to the Clairvoyant algorithm.
3. The rewards obtained considering the true matching rewards distributions.



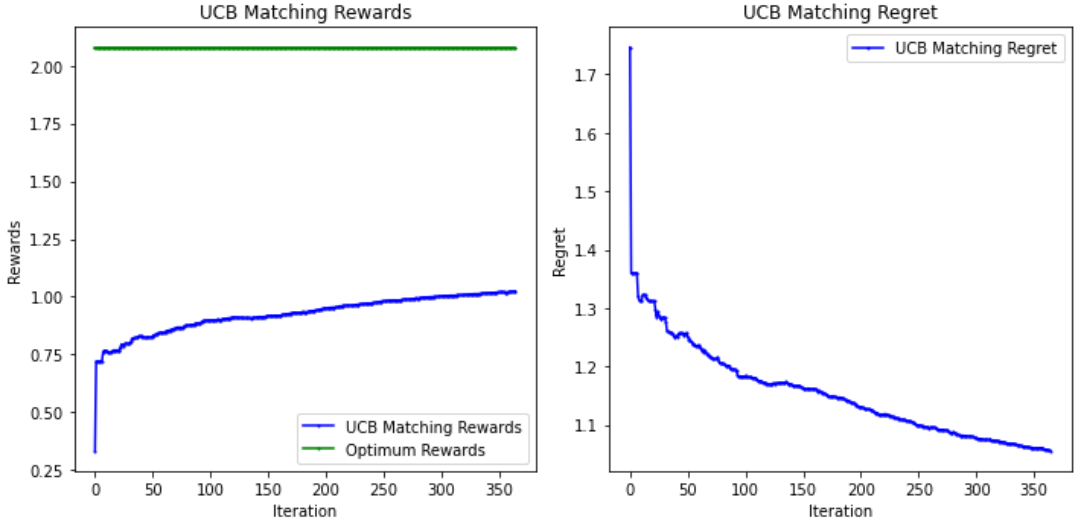




Comparing Matching UCB, TS and Clairvoyant (2/4)

Instantaneous reward and regret over time for **TS Matching** compared to matching with reward distributions known a priori.

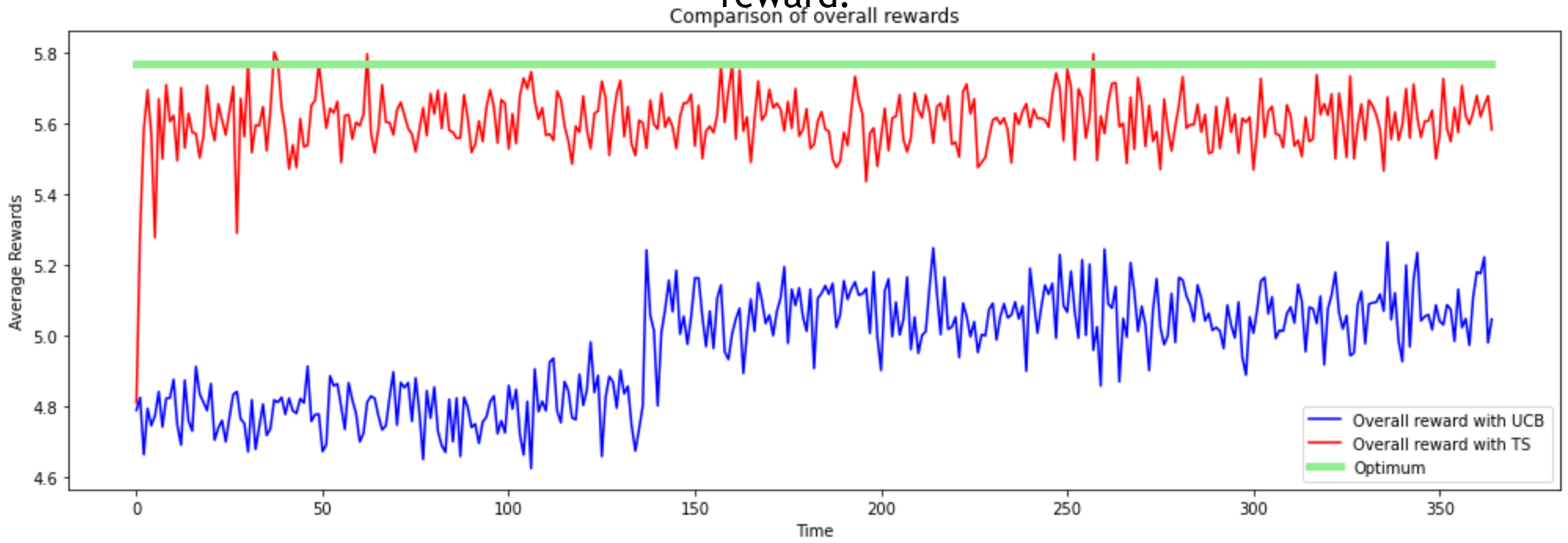
Instantaneous reward and regret over time for **UCB Matching** compared to matching with reward distributions known a priori.



Instantaneous rewards are computed as the sum of the expected values of the optimal matchings.
The faster convergence of TS might be due to the fact that, differently from UCB, in TS we use the Beta means as expectations over the matching reward means, and such Beta means are updated at every round for all arms. Conversely, in UCB we utilized the reward empirical means as expectations, and empirical means are updated only when an arm is played as part of a superarm.

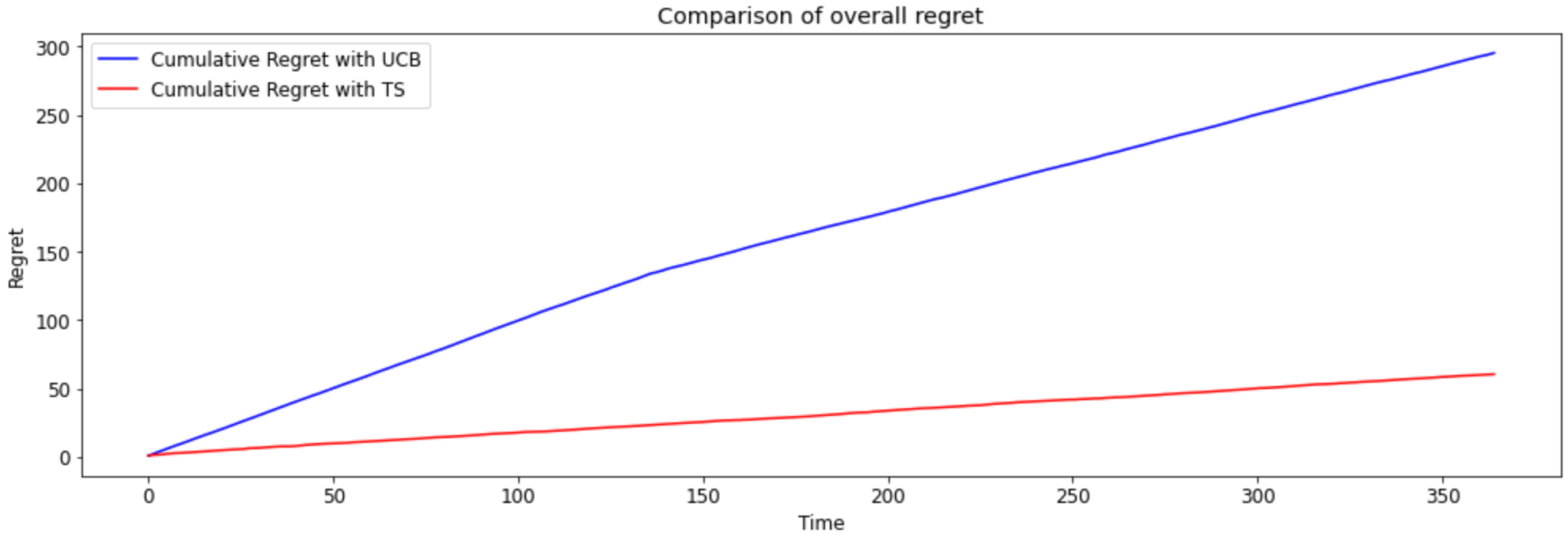
Comparing Matching UCB, TS and Clairvoyant (3/4)

Overall rewards (influence propagation + matching task) when estimating matching rewards with **UCB**, **TS** and when using **real probabilities**. The slowest convergence of UCB Matching leads to a lower overall reward.



Comparing Matching UCB, TS and Clairvoyant (4/4)

The previous setting has generated the following Cumulative Regrets for **TS** and **UCB**.



5
Learning for joint social influence and matching

Joint social influence and matching (1/2)

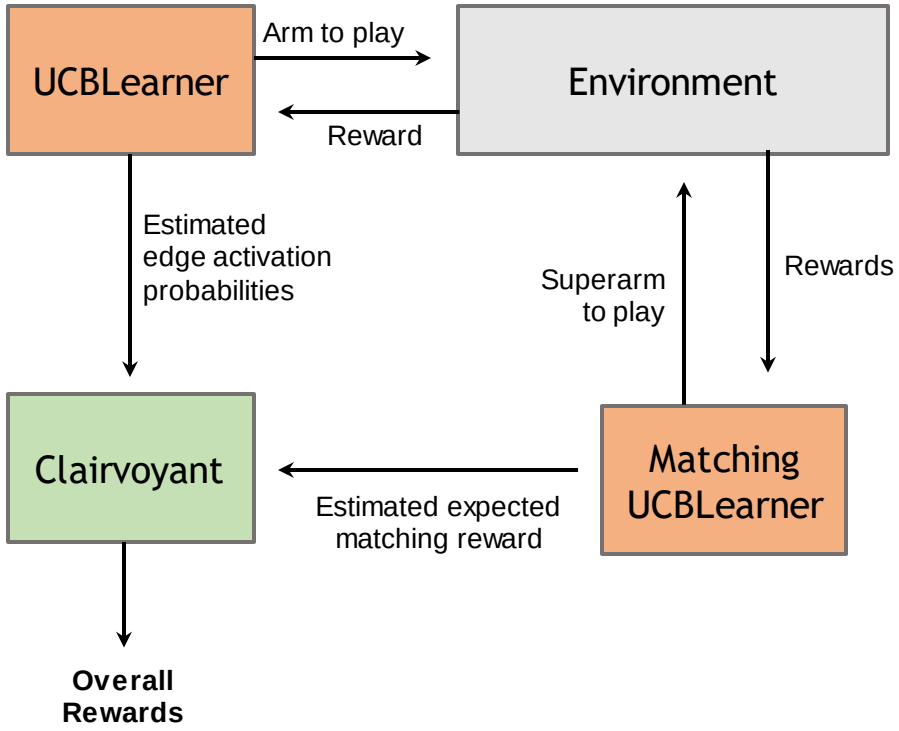
In the previous two sections (Section 3 and 4) we have respectively:
- Used standard bandit algorithms (UCB and TS) to estimate the edge activation probabilities while assuming that the matching rewards were known a priori;
- Used matching bandit algorithms (UCB and TS) to estimate the matching rewards while assuming that the edge activations were known a priori.

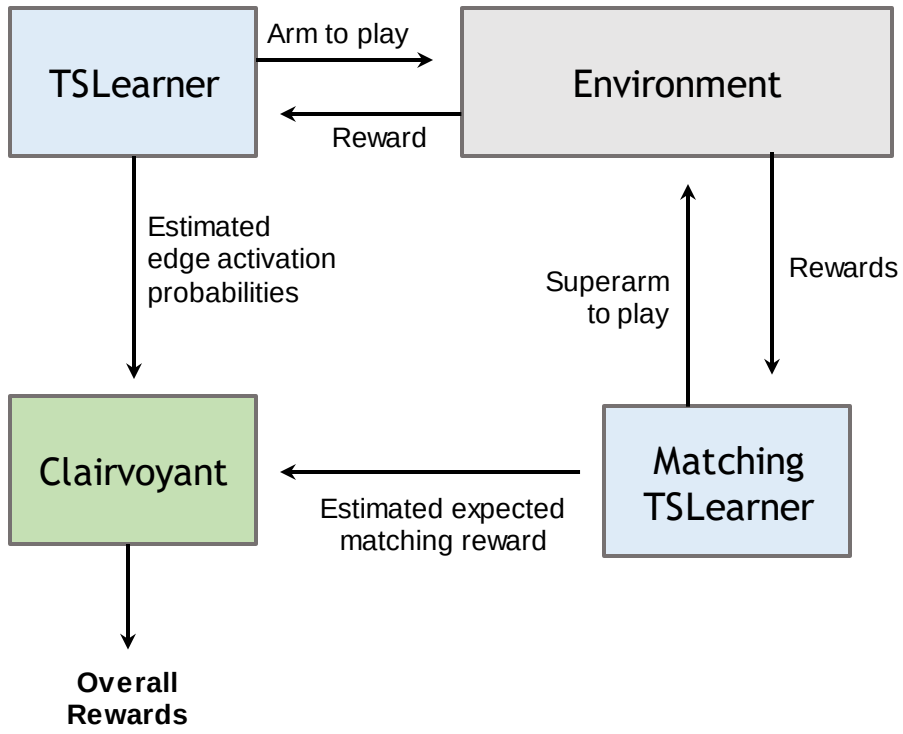
In the current section, we combine the two approaches to deal with fully uncertain settings, in which both the edge activation probabilities and the expected matching rewards need to be estimated. Hence, at each round, we:

- First, produce an estimate of the edge activation probabilities using Standard UCB and TS algorithms.
- Secondly, produce an estimate of the matching reward parameters using Matching UCB and TS algorithms.

The corresponding round estimates are then passed as input to the clairvoyant algorithm that will output the overall generated reward.

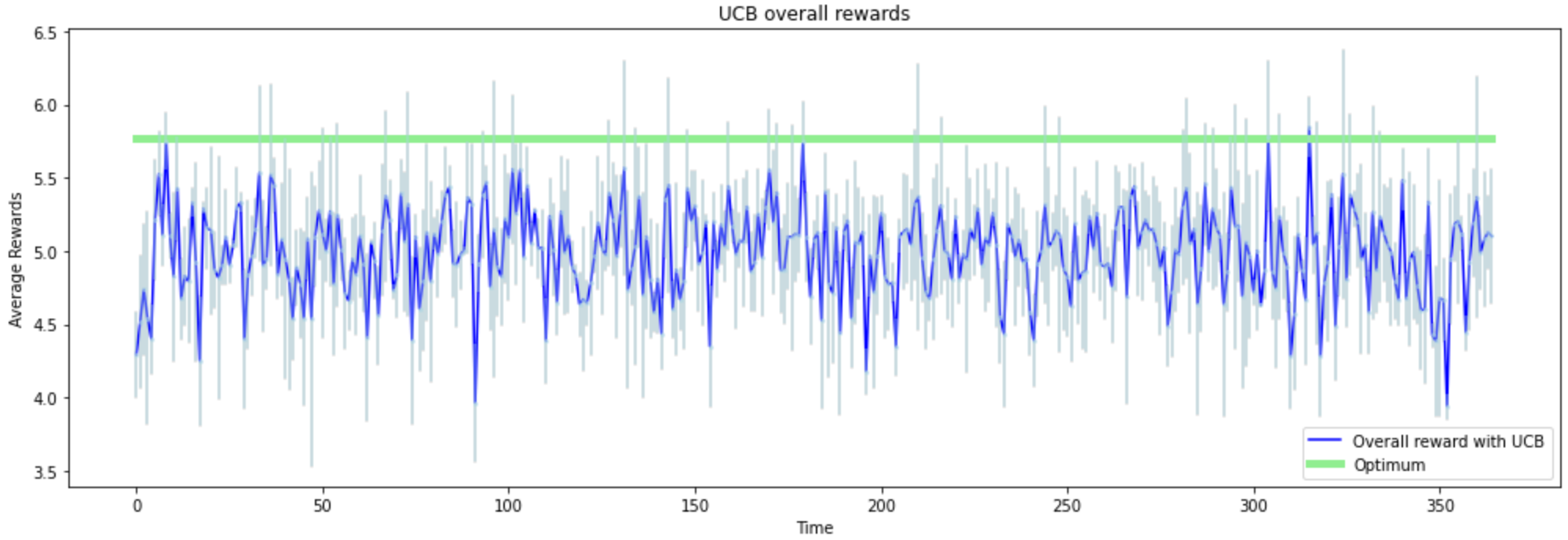
Joint social influence and matching (2/2)





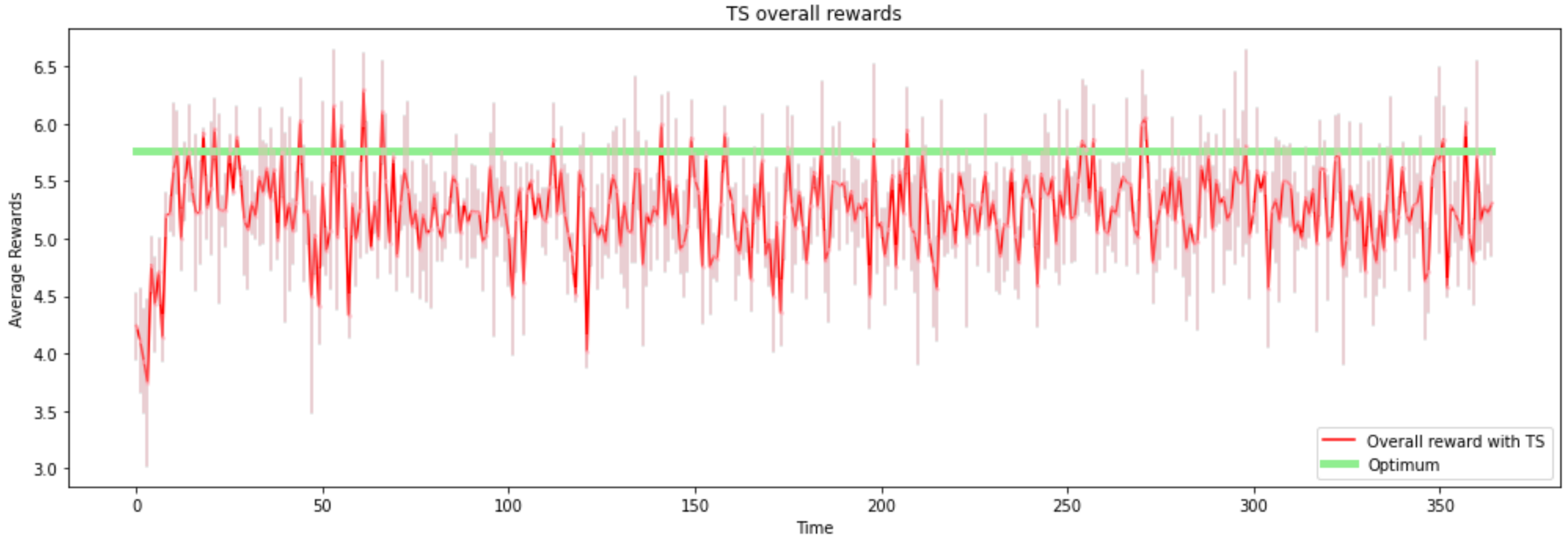
Results of joint social influence and matching (1/4)

The figure shows the means and standard deviations of overall instantaneous reward when using (a) **Standard UCB** to estimate edge activation probabilities and (b) **Matching UCB** to estimate expected matching rewards.



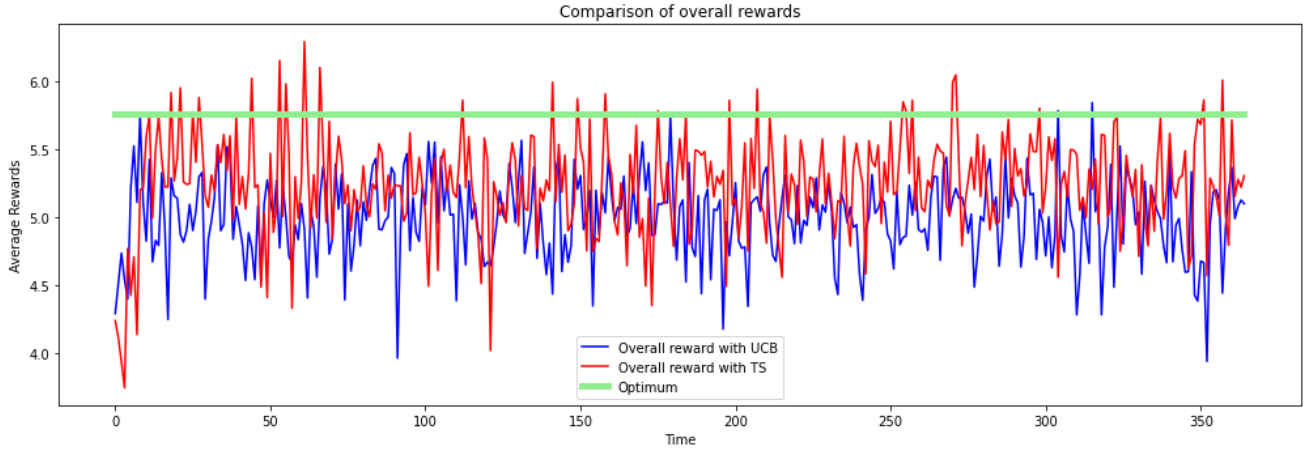
Results of joint social influence and matching (2/4)

The figure shows the means and standard deviations of overall instantaneous reward when using (a) **Standard TS** to estimate edge activation probabilities and (b) **Matching TS** to estimate expected matching rewards.

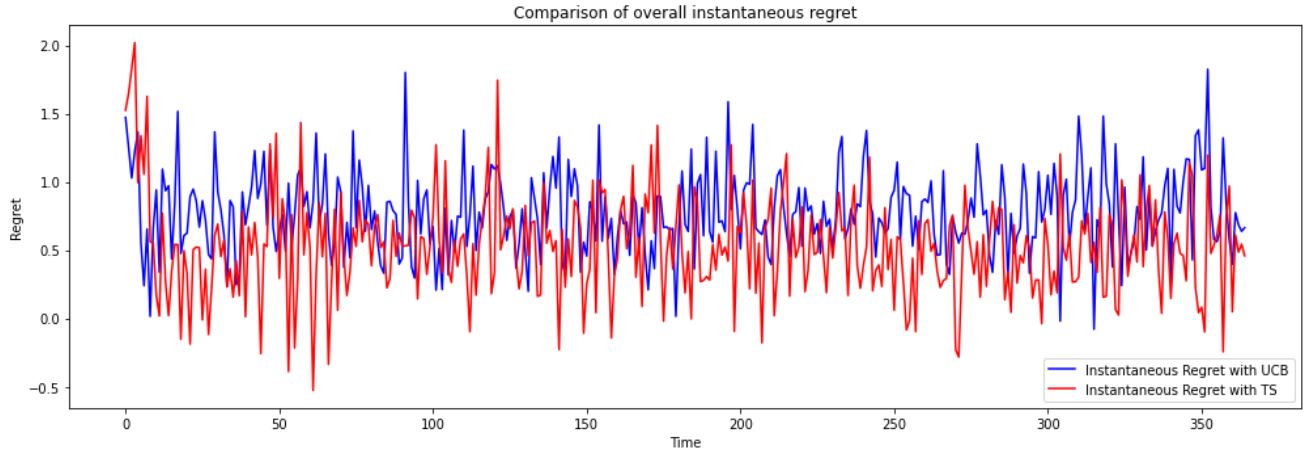


Results of joint social influence and matching (3/4)

Comparison of instantaneous reward means in the two cases.

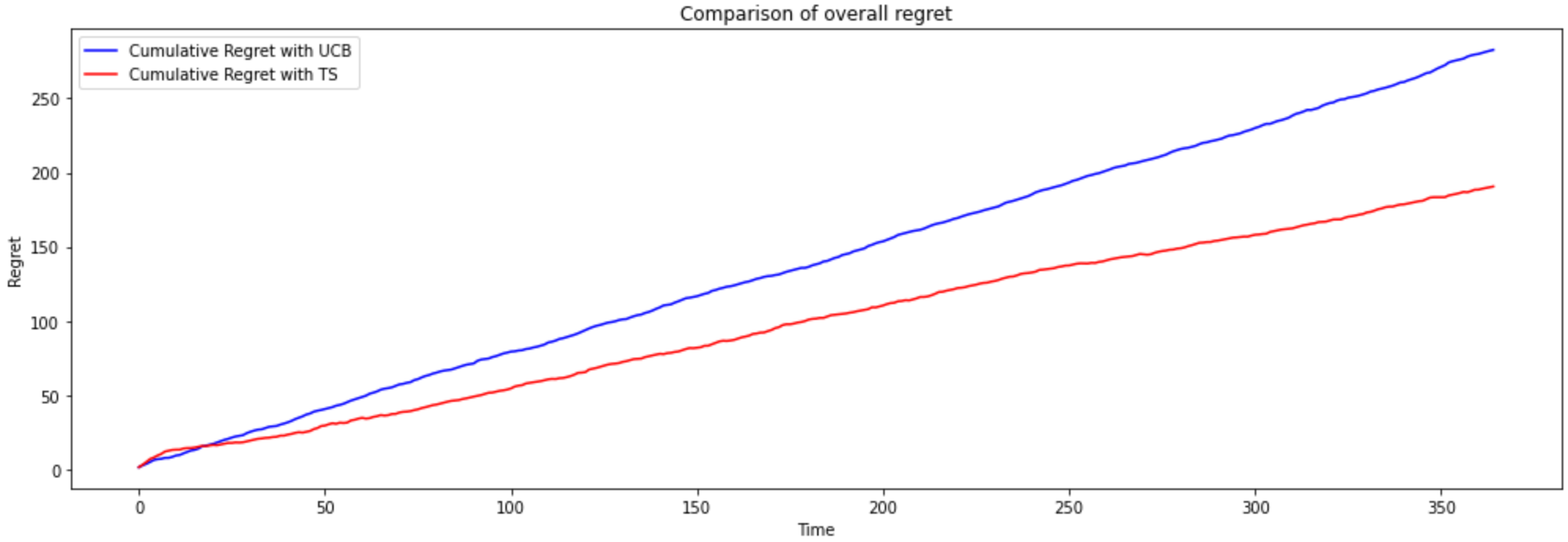


Comparison of instantaneous regrets in the two cases.



Results of joint social influence and matching (4/4)

The following is a comparison of cumulative regrets in the two cases. In both, the cumulative regret appears to be linearly increasing in time.

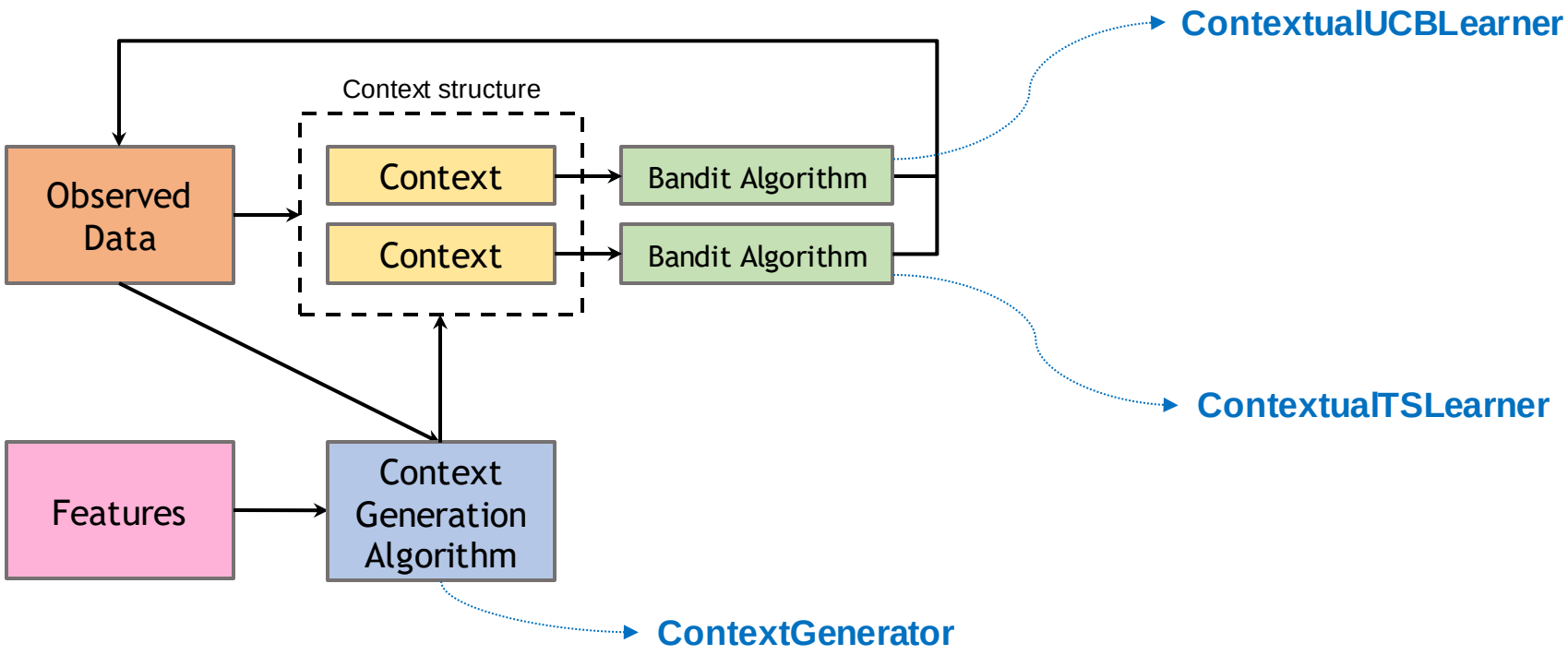


6
Contexts and their generation

Different Contexts (1/1)

INTRODUCTION

In this scenario, the classes of customers are not defined. Moreover, no information about the edge activation probabilities and reward distributions is known. Everything needs to be learned from the observed data. Our learners do not know how many contexts (i.e. classes of customers) there are, they can only observe the two binary features and the data associated to them.



Context Generator (1/1)

PSEUDOCODE

The ***ContextGenerator*** class is responsible for generating contexts based on the features associated with the arms. It uses the *k-means clustering* algorithm to group the features into clusters, which are then used as contexts. Context Generator works following three steps:

1. *Initialization* of the ContextGenerator class with the specified number of clusters (we used n_clusters = 5).
2. *Fitting of* the k-means clustering algorithm to the provided features.
3. *Prediction* of the cluster labels for the provided features based on the trained k-means clustering algorithm.

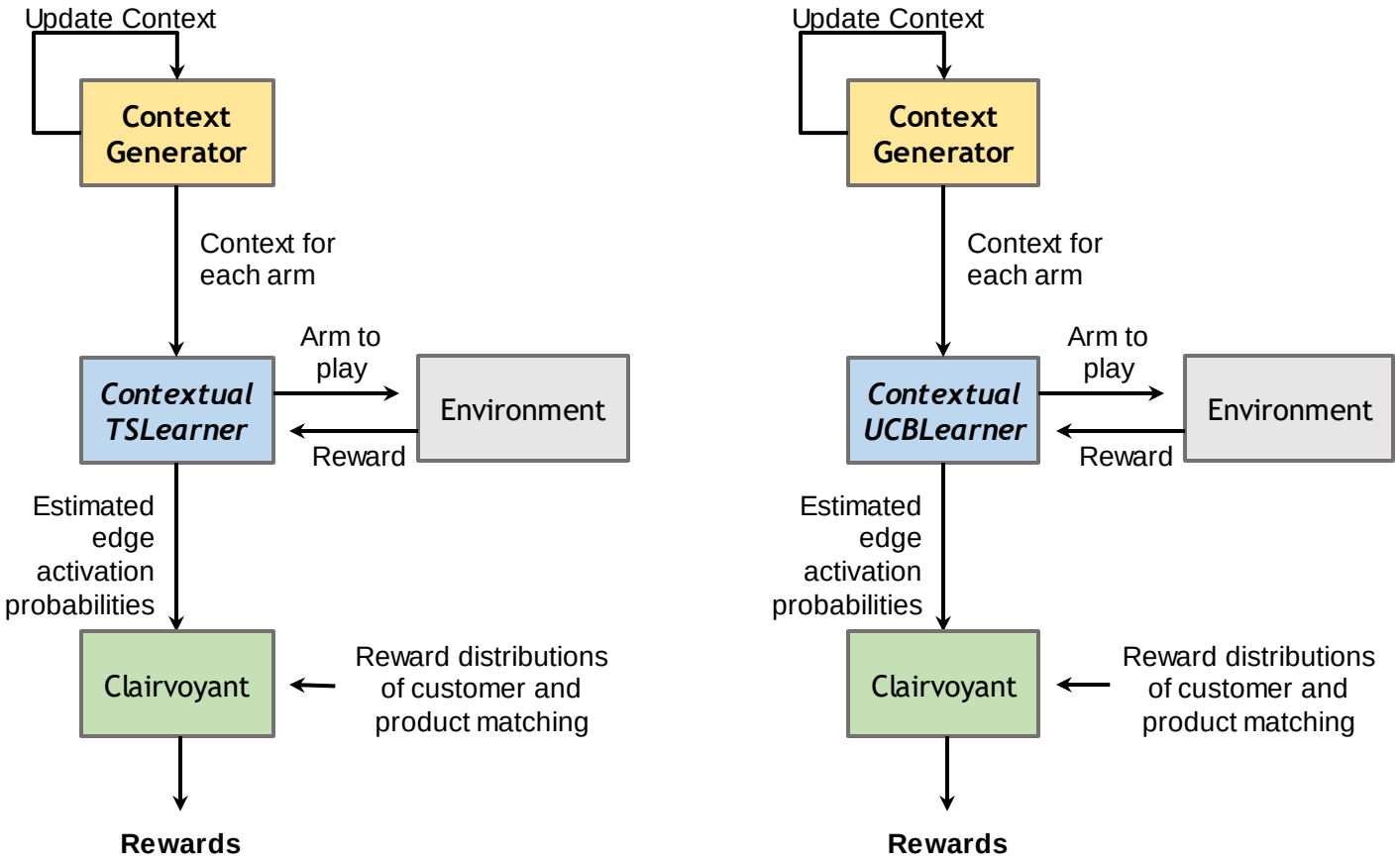
```python
class ContextGenerator:
    def __init__(self, n_clusters):
        self.n_clusters = n_clusters
        self.kmeans = KMeans(n_clusters=n_clusters)

    def fit(self, features):
        self.kmeans.fit(features)

    def predict(self, features):
        return self.kmeans.predict(features)
```

This class possesses the limitations of K-means clustering, specifically:

- Choice of the number of clusters
- Assumption of spherical clusters

Contextual UCB and Contextual TS Learner (1/1)

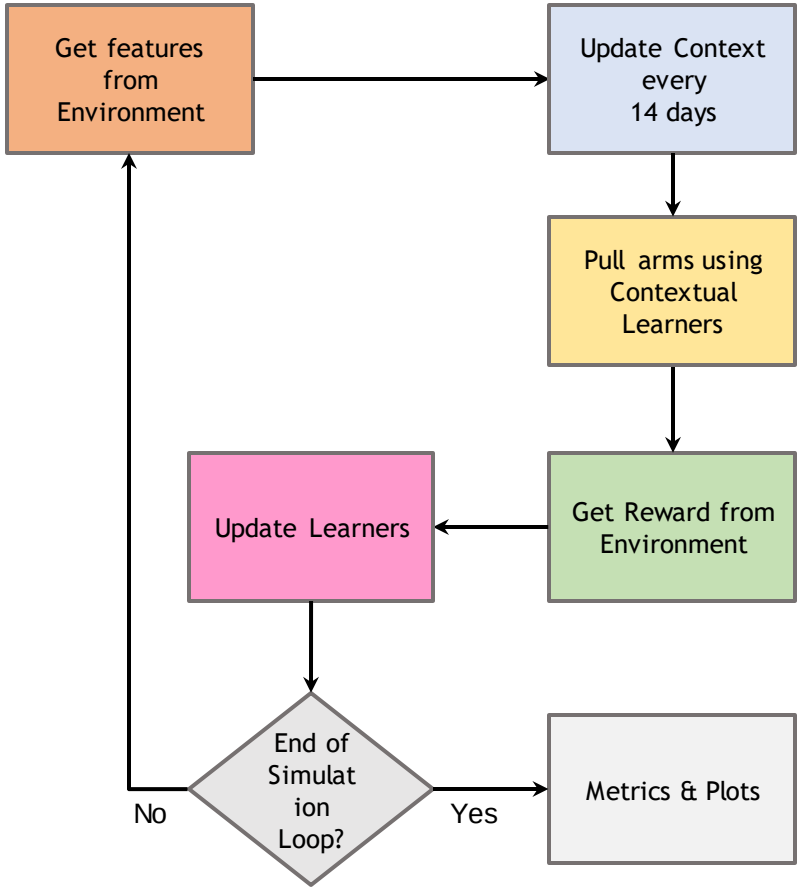
The learners are an updated version of the ones used in step 3. The difference is that they are paired with a context generation algorithm, making them capable of updating the contexts.



Functioning (1/1)

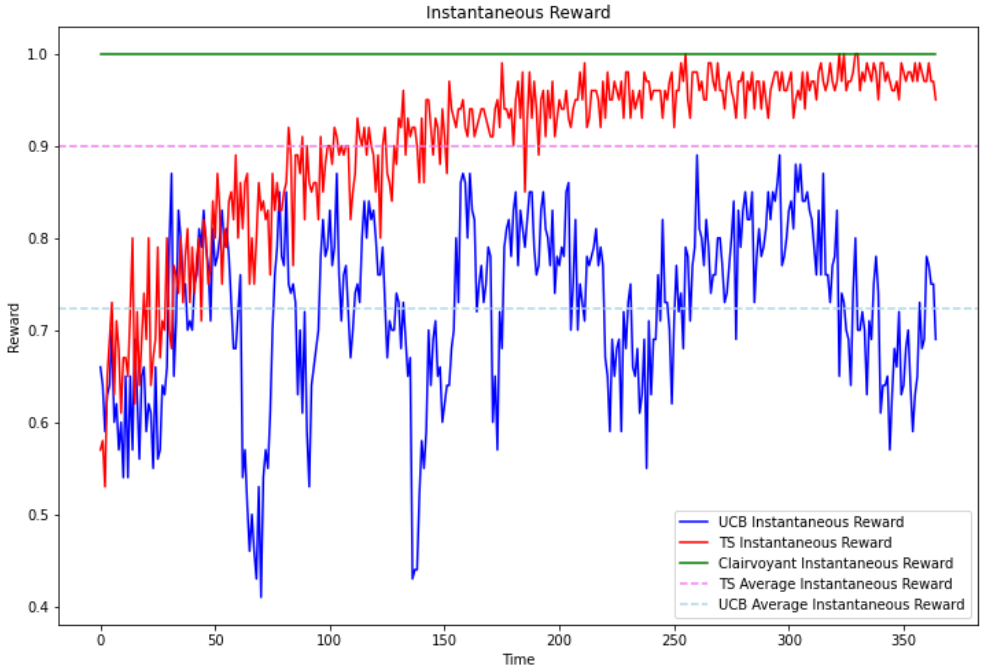
The ***ContextGenerator*** class generates contexts based on the features of the arms, the ***Environment*** class simulates the multi-armed bandit problem, and the ***ContextualUCBLearner*** and ***ContextualTSLearner*** classes implement the UCB and TS algorithms with the ability to handle contexts. The simulation loop runs for a specified time horizon (365 days), updates the contexts, pulls the arms, and updates the learners in each iteration.

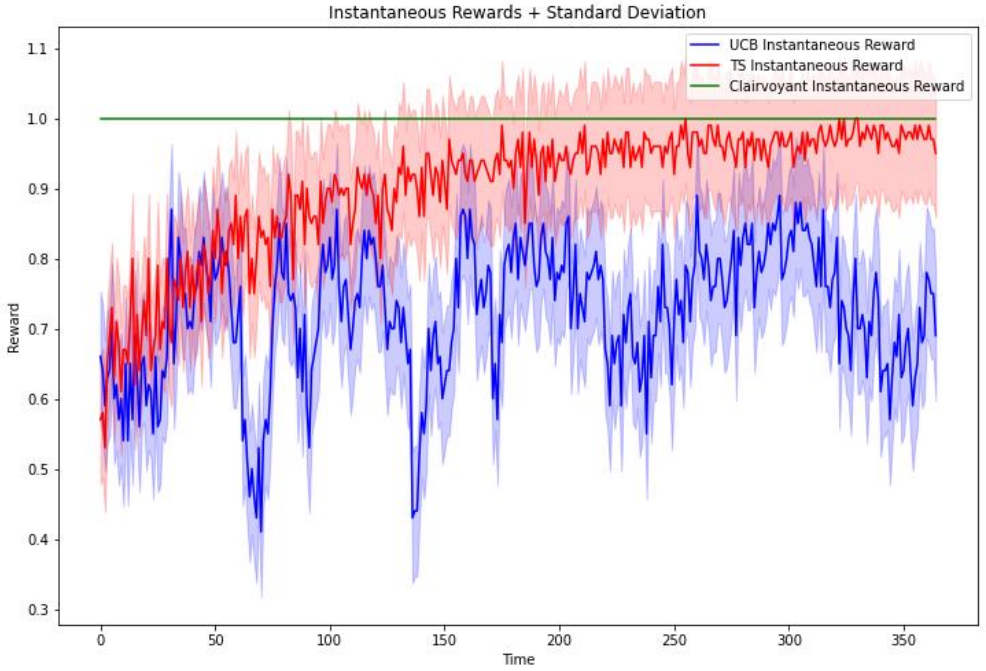
```
UCB Instantaneous Reward: 0.62
UCB Cumulative Reward: 231.92000000000002
UCB Instantaneous Regret: 0.36
UCB Cumulative Regret: 0.36
UCB Standard Deviation: 0.15249762986939033
TS Instantaneous Reward: 0.94
TS Cumulative Reward: 315.8099999999999
TS Instantaneous Regret: 0.04
TS Cumulative Regret: 0.04
TS Standard Deviation: 0.1226276824708144
```



Instantaneous Reward (1/1)

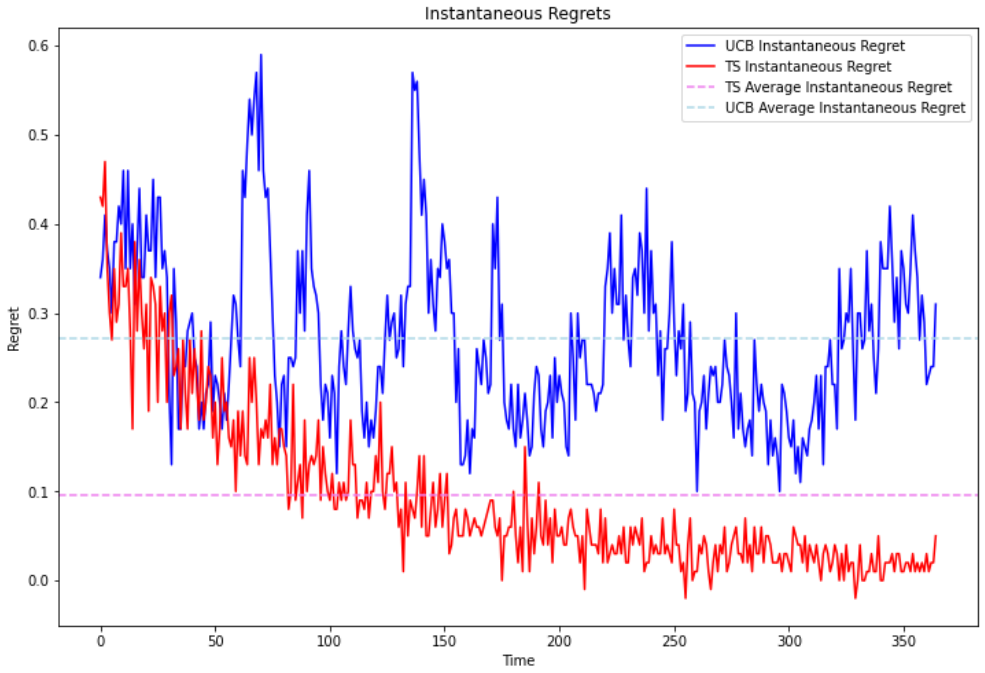
In a setting where the Contexts change every 14 days, the **UCB** has very unpredictable instantaneous rewards, which tend to increase over time in following a very inconsistent trend, while the **TS** instantaneous reward grow more regularly, with smaller peaks and valleys

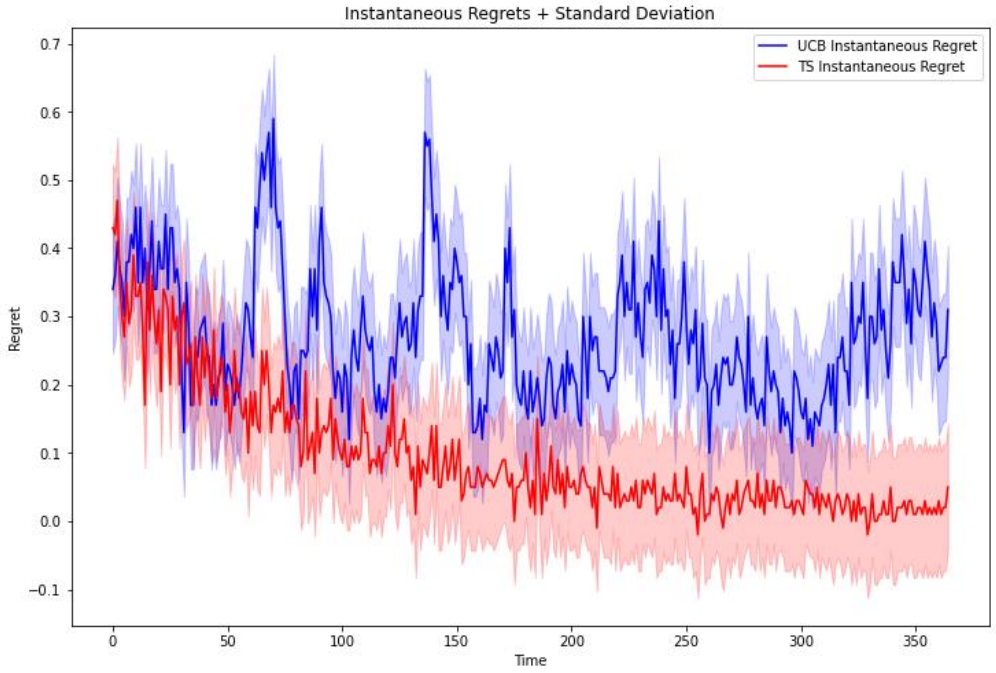




Instantaneous Regret (1/1)

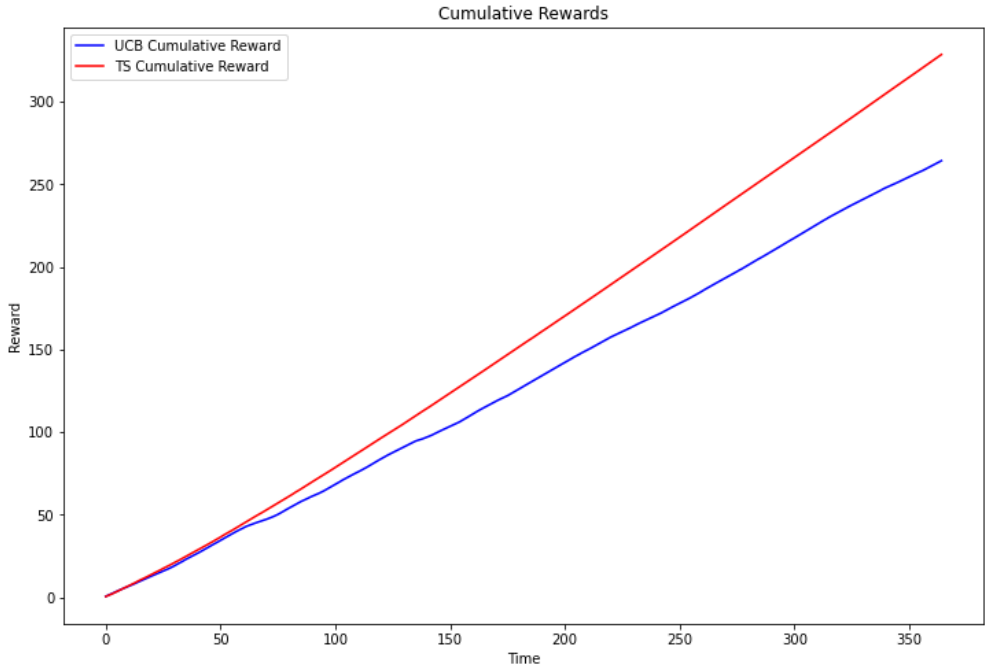
In a setting where the Contexts change every 14 days, the **UCB** has higher average instantaneous regrets, which slightly decrease over time in following a very inconsistent trend, while the **TS** instantaneous regret follows a decreasing trend that is more regular, with smaller peaks and valleys. The average regret value for the **TS** algorithm is smaller than the one of the **UCB**.





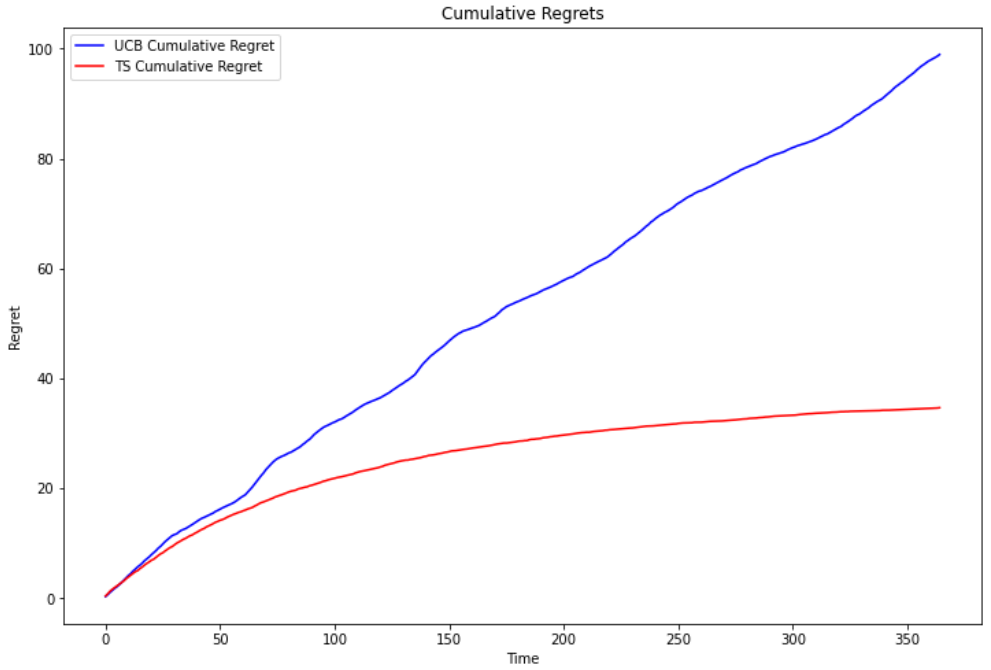
Cumulative Reward

In a setting where the Contexts change every 14 days, the **TS** cumulative reward is significantly higher than the **UCB** cumulative reward



Cumulative Regret

In a setting where the Contexts change every 14 days, the **TS** cumulative regret is significantly smaller than the **UCB** cumulative regret



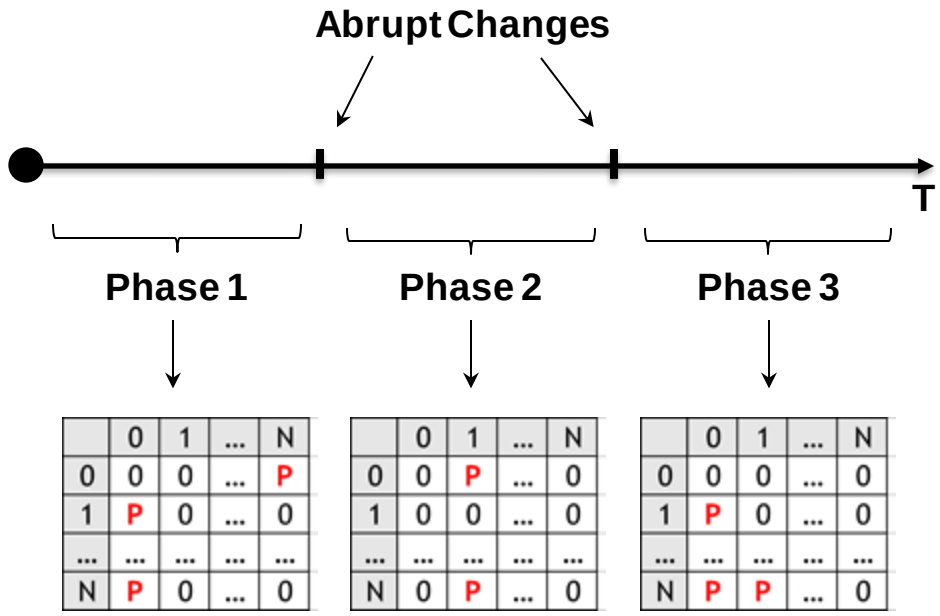
7
Non-stationary environments with two abrupt changes

Hypothesis of non-stationarity (1/1)

We assume that:

- Edge activation probabilities are **unknown** and **non-stationary**. They are subject to **two abrupt changes dividing the time horizon into three equally long phases**.
- All other properties of the social influence setting are known. The optimal customer-product matching is also known.

Realistically, these conditions suggest that customers might suddenly and significantly change the way they influence each other depending on the current phase. This could be motivated by **seasonal patterns in the social influence dynamics**. For instance, as previously discussed in Section 1, we could assume that the way customers influence each others' purchases during the holiday season changes with respect to other moments of the year.



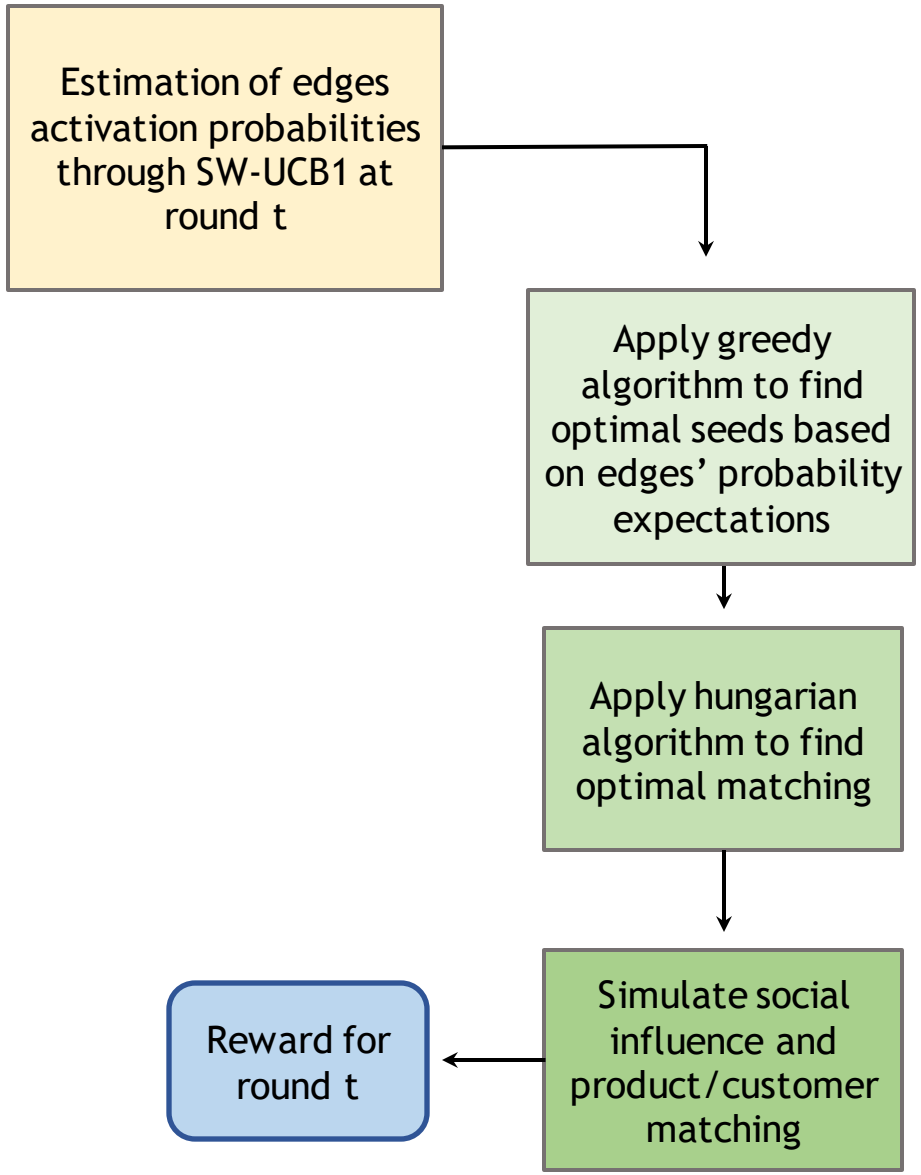
Sliding Window UCB1 (1/3)

A first proposed solution involves progressively estimating the edge probabilities over a fixed sliding window of time. In particular, using a SW-UCB1 algorithm:

- we model each edge as an arm,
- we progressively compute its expected activation probability as the mean of its Bernoulli rewards observed in the last time window.

At each round:

1. The edge activation probabilities are estimated as explained above.
2. The clairvoyant algorithm is applied to find the 3 optimal seeds, simulate the influence propagation, and finally assign products.



Sliding Window UCB1 (2/3)

Here is a brief review of **SW-UCB1** mechanism:

- The algorithm keeps track of a list of played arms at each round. Each arm has its own array of collected rewards.
- Each arm is played once in the first round. While the number of rounds is smaller than the time window size τ, the algorithm functions as a standard UCB1. At each round, the algorithm outputs the mean of the rewards collected for each arm.
- Once the time passed exceeds the window size, at each round:
  - If there are arms that were not played in the previous time window, SW-UCB1 chooses randomly which arm to play among them. To check if an arm has been played, the algorithm searches for it in the last τ items in the list of played arms.
  - Otherwise, the algorithm pulls the arm with the highest confidence bound.
  - Once the arm has been played, SW-UCB1 updates its mean reward and confidence bound. Both mean and confidence bound are computed only by taking in consideration the last τ rounds.
  - The algorithm outputs the current state of the means of rewards in the last τ rounds to be used as expectations of edge probabilities.

Sliding Window UCB1 (3/3)

As a demonstration, assuming that the window size is $\tau = 4$, and that in the current round the list of played arms is:

[…, …, …, 10, 11, 9, 20, 11]

Each arm has an array storing the rewards obtained at each pull:

…
Arm 9 : […, …, …, 0.33]
Arm 10 : […, …, …, 0.12]
…
Arm 11 : […, 0.81, 0.64, 0.57, 0.93, 0.86]
…
Arm 20: […, …, …, 0.55]
…

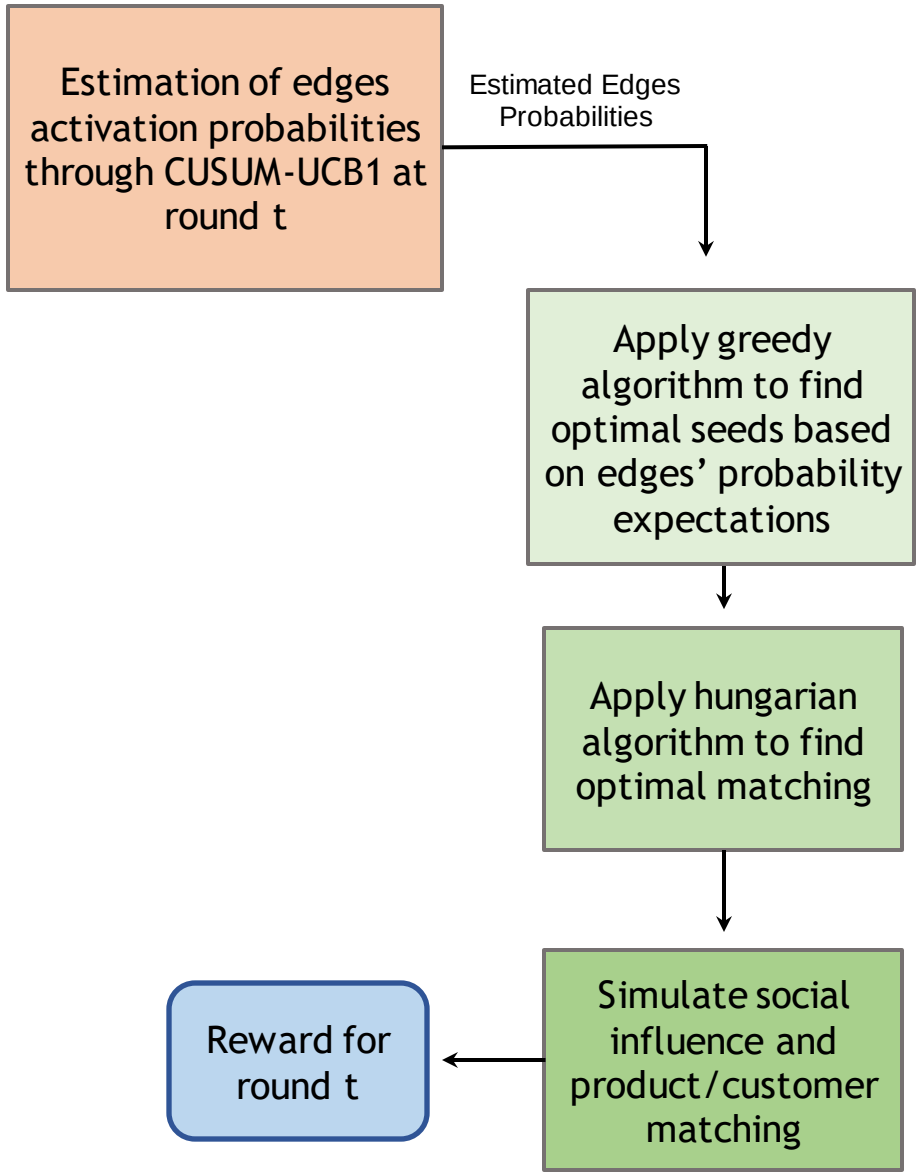
- To check if arm 9 was played in the last time window, the algorithm checks for the presence of 9 in the last $\tau = 4$ elements of the list of played arms.
  [11, 9, 20, 11]

- To compute the mean reward of arm 11 in τ, the algorithm counts how many times the arm occurs in the last $\tau = 4$ elements of the list of played arms. In our case, 11 appears twice.
  [11, 9, 20, 11]
  Hence, the algorithm computes the mean over the last two rewards stored for arm 11.

Change Detection CUSUM-UCB1 (1/4)

A second solution involves using a change detection mechanism. Also in this case, edge probabilities are estimated as the mean of rewards obtained in a UCB1 process over a time window. The setting is almost the same as before. The only difference is that each arm has its own individual time window, which is zeroed (reset) every time an abrupt change in the reward distribution of the arm is detected.

At each round:
1. The edge activation probabilities are estimated as explained above.
2. The clairvoyant algorithm is applied to find the 3 optimal seeds, simulate the influence propagation, and finally assign products.

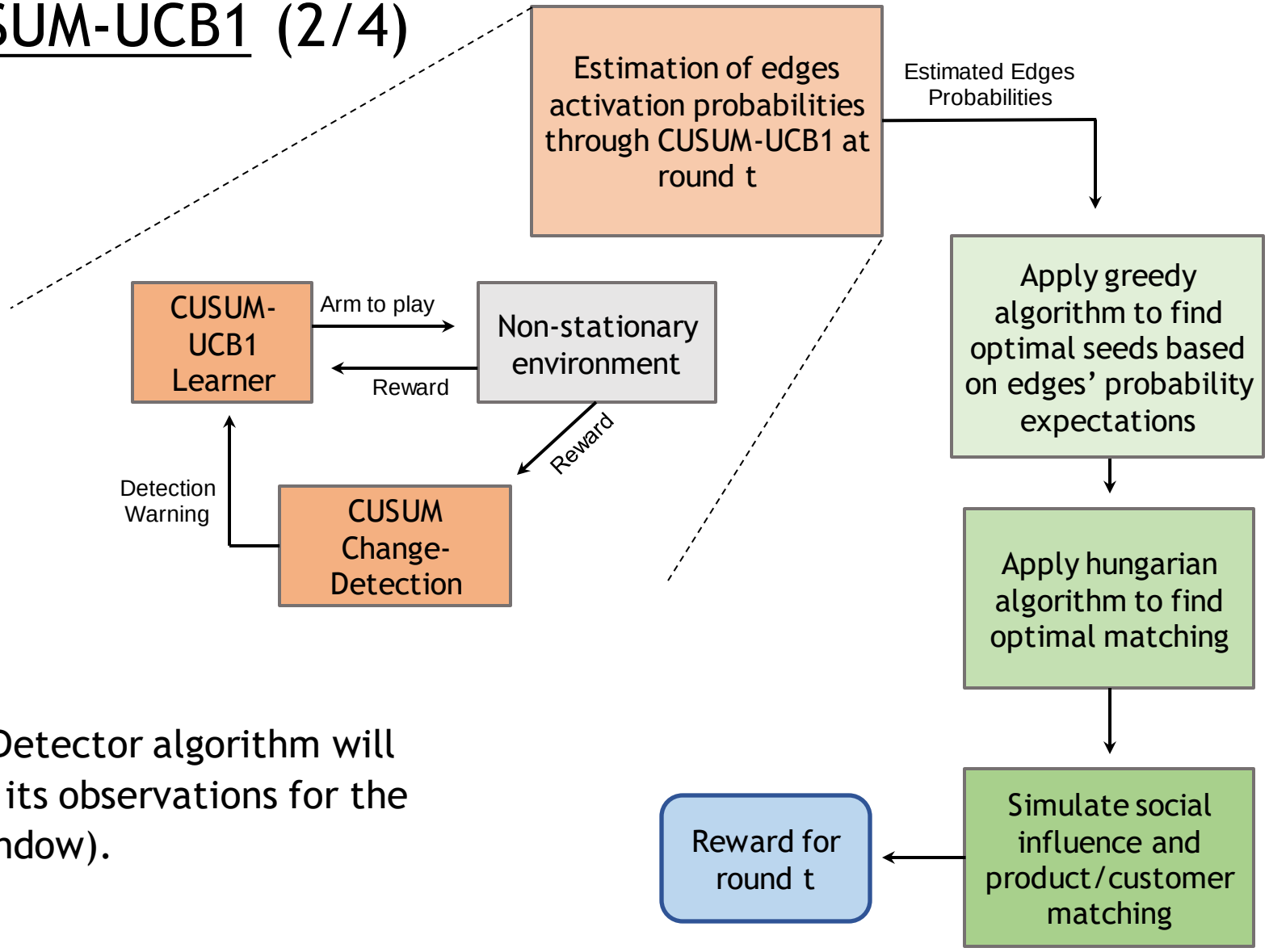


Change Detection CUSUM-UCB1 (2/4)

The UCB1 learner receives the rewards from the non-stationary environment and consequently selects the arm that will be played.

A Change Detector also receives the rewards from the non-stationary environment and utilizes them to detect abrupt changes in the rewards distributions, for each arm.

In case of detection, the Change Detector algorithm will instruct the learner class to reset its observations for the arm (i.e. to zero the arm time window).



Change Detection CUSUM-UCB1 (3/4)

Here is a brief review of **CUSUM-UCB1** mechanism:

- For each arm, the time window τ_a is initialized as 0. The parameter α is defined, which is the probability that a random arm is played instead of an arm with the highest confidence bound.
- At each round:
  - With probability α, the algorithm plays a random arm and memorizes the obtained reward. With probability 1 – α, the algorithm plays the arm with the highest confidence bound.
  - The algorithm then updates mean and confidence bound of the played arm a. Such mean and confidence bound are computed only with respect to the last τ_a rewards.
  - The algorithm outputs the current state of the means of rewards in the last τ rounds to be used as expectations of edge probabilities.
  - The Change Detection system for arm a (CD_a) is updated.
    - If the CD_a system returns 1, the arm's time window is set to zero: $\tau_a = 0$.
    - Else, τ_a = t

Change Detection CUSUM-UCB1 (4/4)

The change detection algorithm deployed is the **Cumulative Sum algorithm (CUSUM).** A CUSUM detection system is associated to every arm (i.e. every edge activation probability). This is the pseudocode of Cumulative Sum of arm *a*: CUSUM_a.

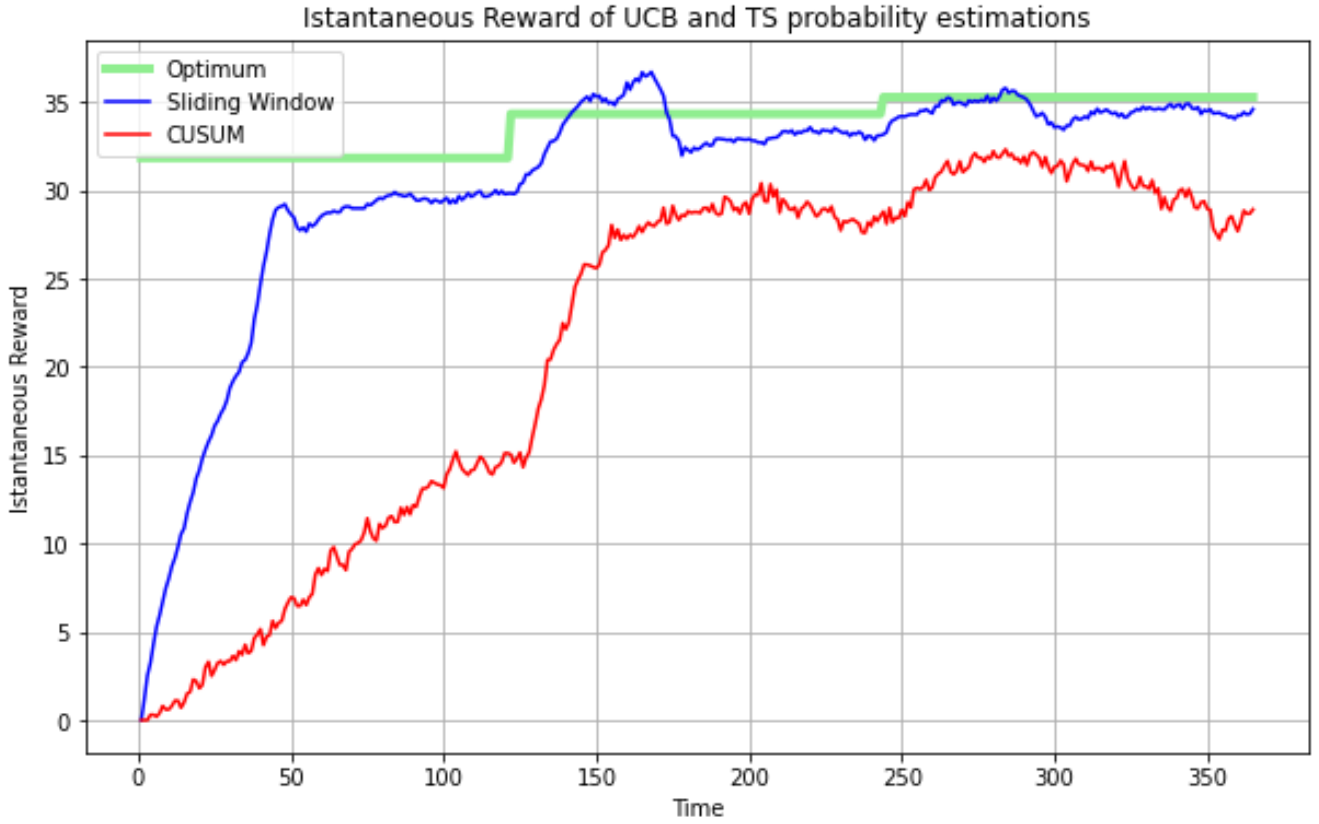
- The following parameters are initialized:
  - M = number of samples needed to produce a "reference mean" of rewards
  - ε = parameter that can be used to adjust the sensitivity of the detector (the larger ε, the less sensitive CUSUM becomes).
  - h = threshold for a change to determine a detection warning
- When the samples of *a* are equal to M, at each round the algorithm checks if there is a detection by:
  1) Computing the positive and negative deviations of the new reward from the reference mean minus the parameter ε.
  2) Computing the cumulative sum of (past and current) absolute values of the positive and negative deviations. The cumulative sums have a lower bound = 0.
  3) Checking whether the positive or negative cumulative sums exceeds the threshold h.
- If the previous condition occurs, the algorithm returns True, signaling a detection of abrupt change for the reward distribution of the arm *a*.

Testing (1/7)

The estimation of non-stationary edge activation probabilities through the two algorithms led to the following instantaneous rewards over the time horizon. Instantaneous rewards are computed as the sum of the estimated activation probabilities in the round.

- Sliding Window UCB
  Window size = int(T/10)

- Change Detection CUSUM UCB
  M = 2, ε = 0.01, h = 0.3, α = 0.2

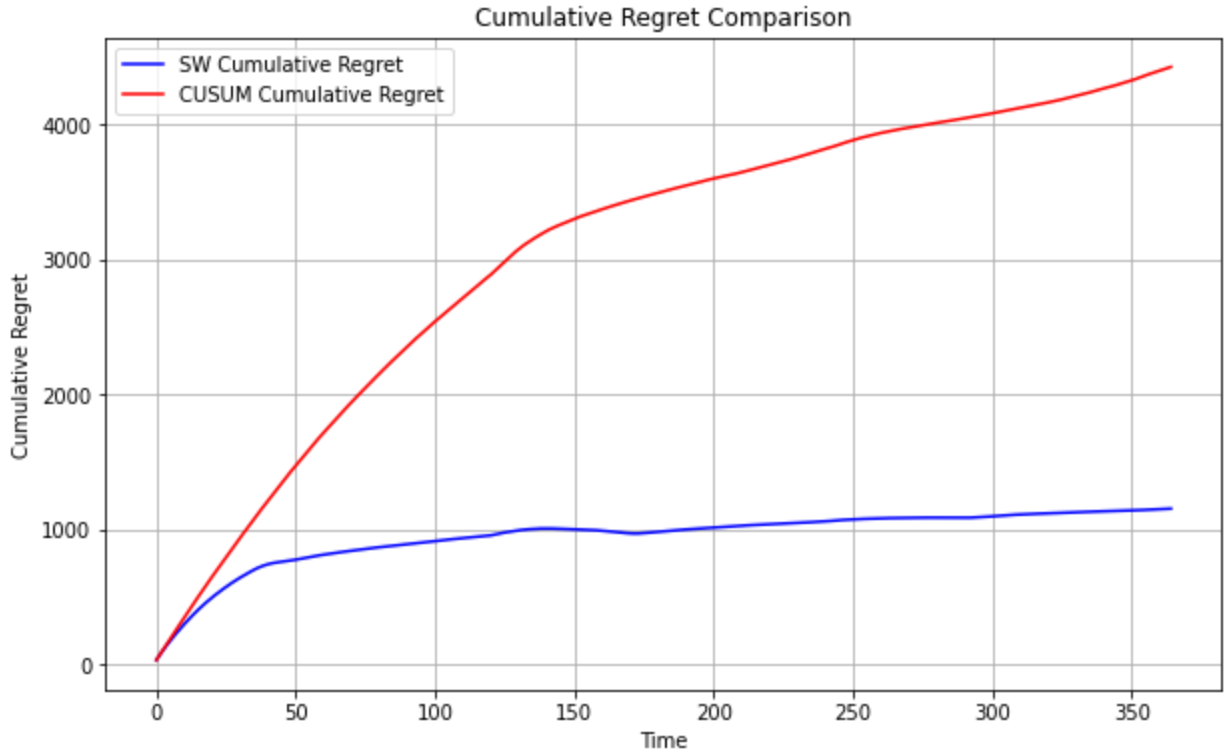


Testing (2/7)

By comparing the two cumulative regrets, Sliding Window UCB appears to outperform the Change Detection UCB learner in the probability estimation task. Both cumulative regrets increase sub-linearly over time.

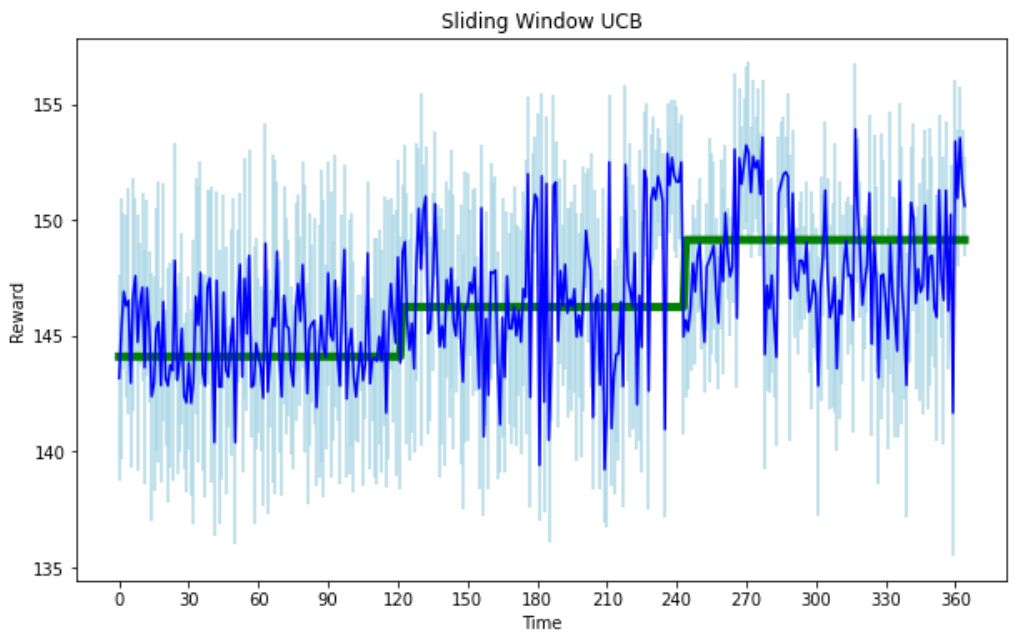
- Sliding Window UCB
  Window size = int(T/10)

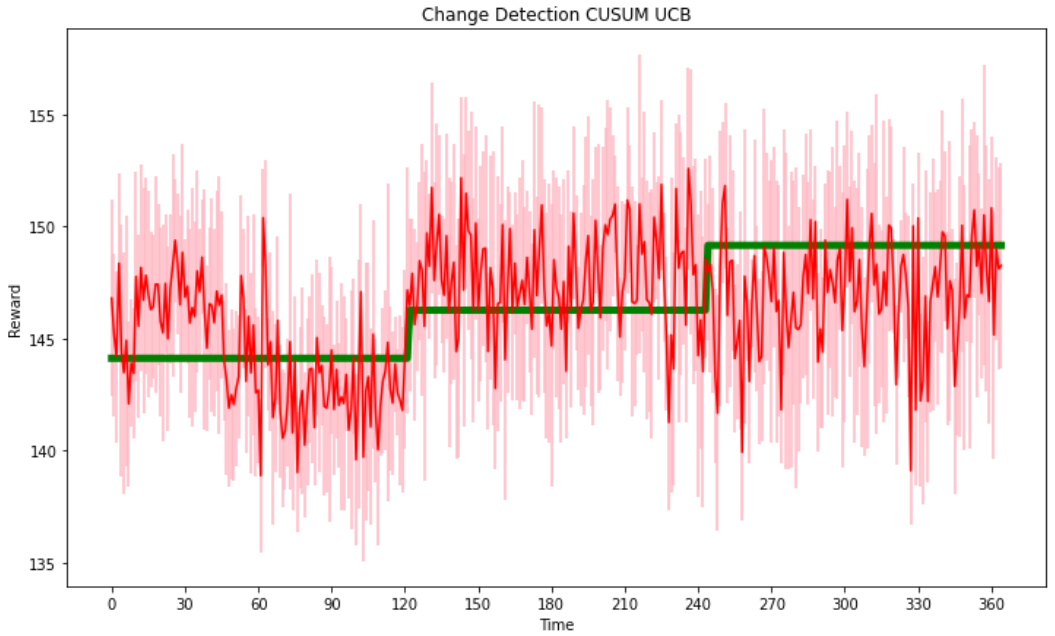
- Change Detection CUSUM UCB
  M = 2, ε = 0.01, h = 0.3, α = 0.2



Testing (3/7)

Assuming the implementation is correct, the two algorithms led to the following rewards over the time horizon.





Sliding Window UCB

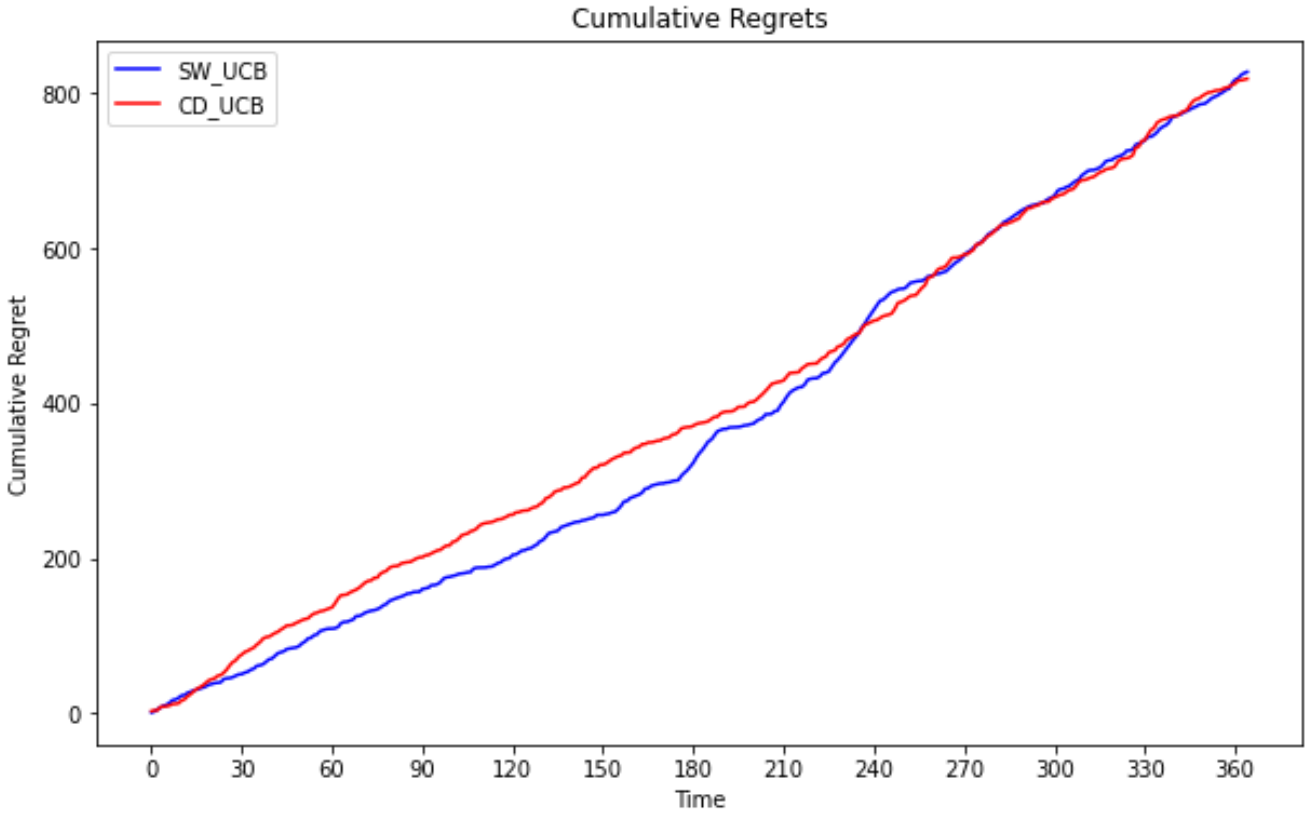
Window size = int(T/6)

Change Detection CUSUM UCB

M = 2, ε = 0.1, h = 0.4, α = 0.2

Testing (4/7)

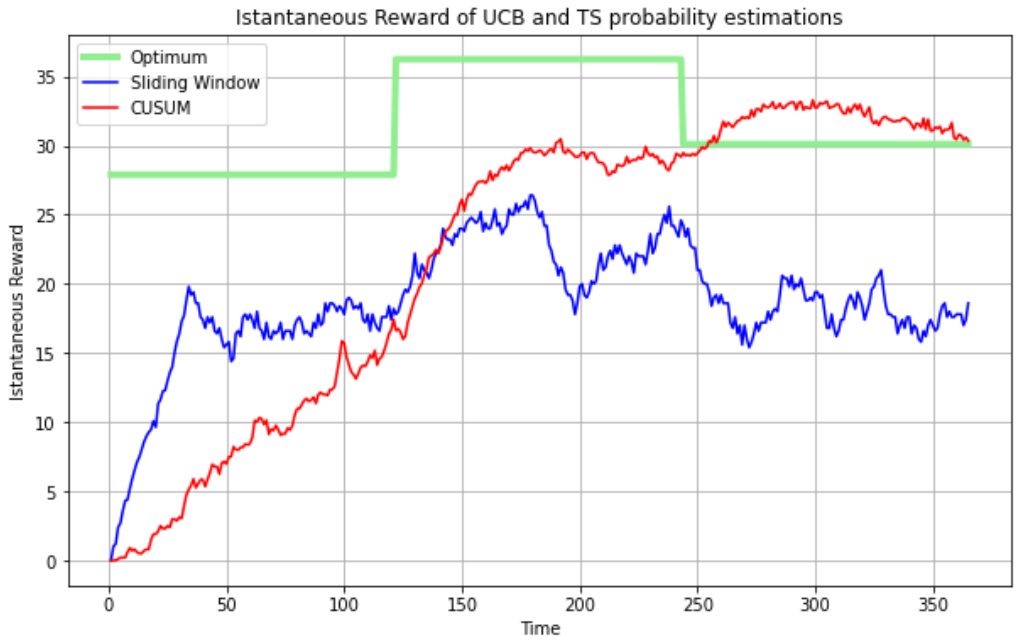
The figure displays a comparison between the cumulative regrets obtained by the two algorithms. Once again, the overall task appears to have a cumulative regret increasing linearly over time.

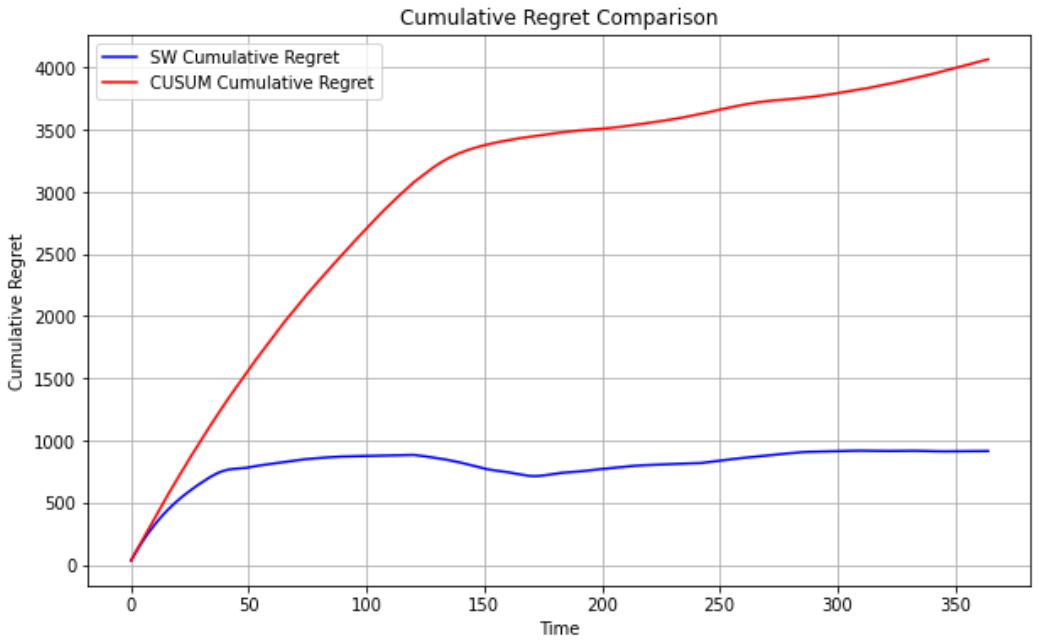


Testing (5/7)

We also conducted other tests, to assess the different performances while changing the algorithms parameters:

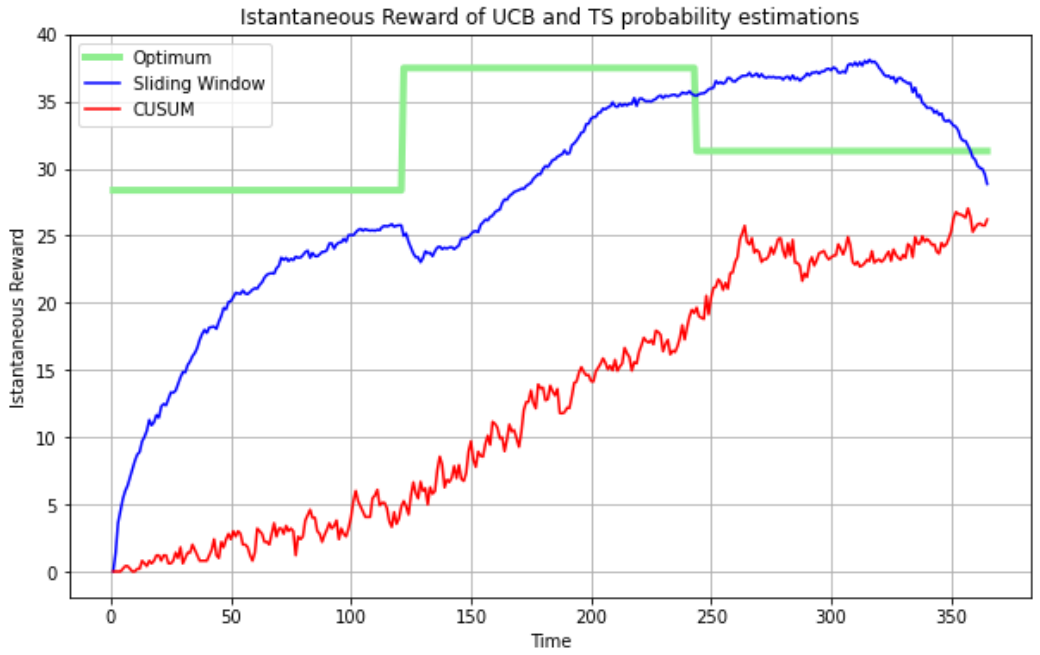
- **Sliding Window UCB**: Window size = int(T**0.5)
- **CUSUM UCB**: M = 2, ε = 0.1, h = 0.6, α = 0.1





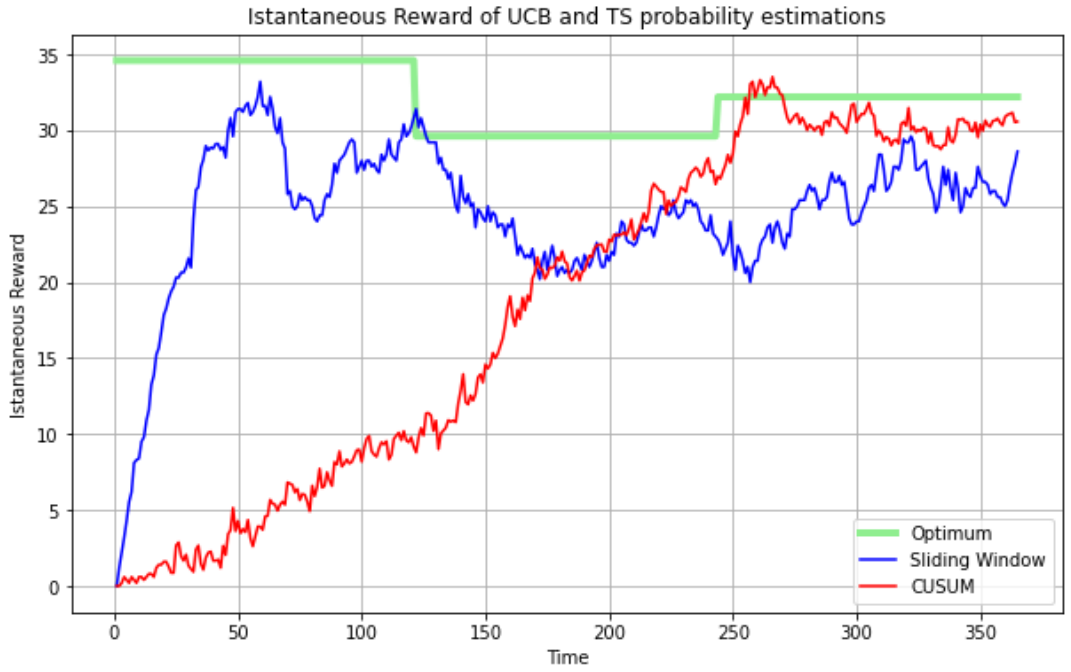
Testing (6/7)

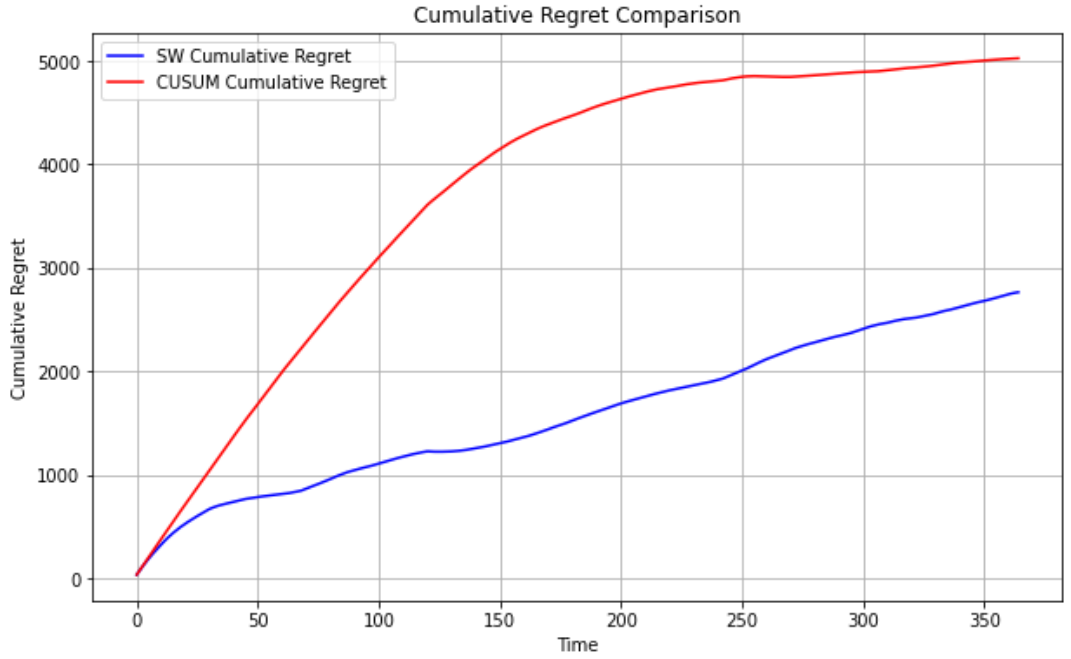
- **Sliding Window UCB**: Window size = int(T/3)
- **CUSUM UCB**: M = 5, ε = 0.1, h = 1, α = 0.1



Testing (7/7)

- **Sliding Window UCB**: Window size = int(T/15)
- **CUSUM UCB**: M = 2, ε = 0.5, h = 0.7, α = 0.1



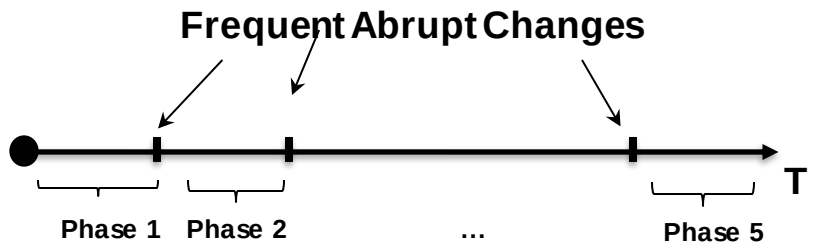


8

Non-stationary environments with many abrupt changes

EXP3 and many abrupt changes (1/3)

The EXP3 algorithm is specifically designed to deal with adversarial settings when no information about the non-stationarity is known beforehand.



- **Finite Number of Arms:** The algorithm assumes a finite number of arms, and it maintains a weight for each arm. The weights are updated based on the observed rewards.

We need to consider the:

- **Exploration-Exploitation Trade-off**: The algorithm balances exploration and exploitation. The exploration-exploitation trade-off is controlled by a parameter called the “exploration parameter” (*gamma*).

EXP3 and many abrupt changes (2/3)

EXP3 PSEUDOCODE:

- The algorithm receives in input a parameter gamma $\gamma \in (0,1]$ and assigns weights wi (initialized as 1) to each arm $i = \{1, \dots, K\}$.
- At each time step, the algorithm:
  - For each arm, computes a probability based on the parameter gamma γ and the corresponding weights w. The probability for the arm i in round t is computed as:

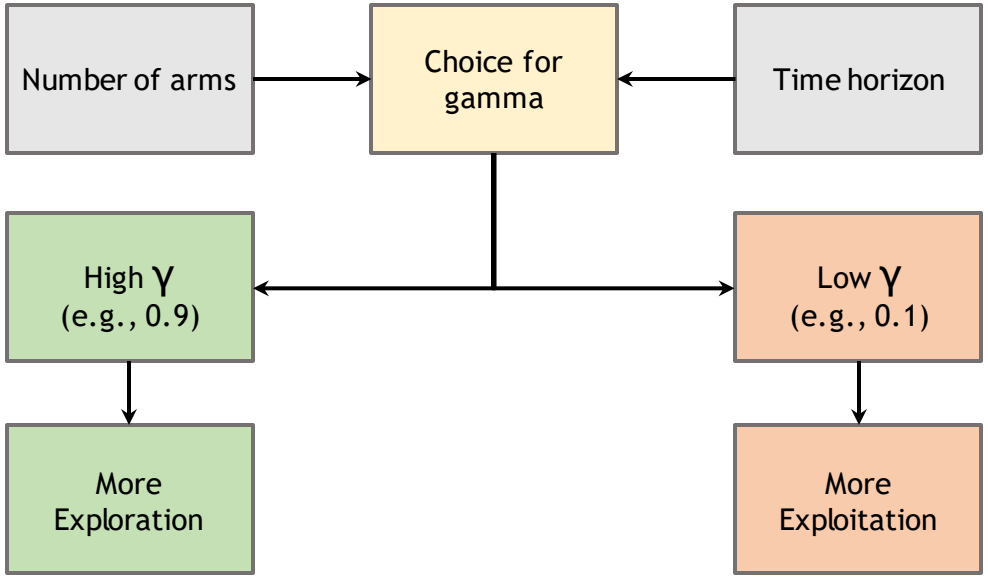
$$p_i(t) = (1-\gamma)\frac{w_i(t)}{\sum_{j=1}^{K} w_j(t)} + \frac{\gamma}{K}, \text{ for } i = \{1, \dots, K\}$$

  - Extracts the arm a_t to play depending on the obtained probabilities and obtains a Bernoulli reward r.
  - Updates the weights of each arm: $w_i(t+1) = w_i(t)\exp\left(\gamma\frac{\hat{r}_i(t)}{K}\right)$ where $\begin{cases}\hat{r}_i(t) = r_i(t)/p_i(t), \text{ for } i = a_t \\ \hat{r}_i(t) = 0, \text{for } i \neq a_t\end{cases}$
  - Computes the empirical means of the Bernoulli rewards and uses them as estimates of the arms rewards (i.e. edge activation probabilities) for the round.

EXP3 and many abrupt changes (3/3)

The best value for the *gamma* parameter in the EXP3 algorithm depends on the specific problem and the characteristics of the environment:

- **Number of arms**: in general, if there are more arms in the problem, it may be beneficial to set a higher value of gamma to encourage more exploration and increase the probability of finding the best arm.
- **Time horizon**: if the time horizon is long, it may be beneficial to set a higher value of gamma to encourage more exploration and increase the probability of finding the best arm over the long term.

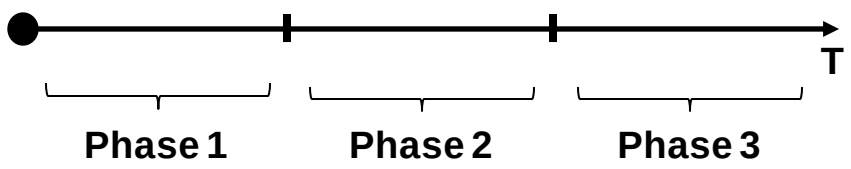


- A higher γ value is beneficial in non-stationary environments where the reward distributions change over time. However, it may also lead to suboptimal performance in the short term, as the algorithm may pull arms with lower expected rewards more frequently.
- A lower may lead to suboptimal performance in the long term, as the algorithm may get stuck in a local optimum and miss out on better-performing arms.

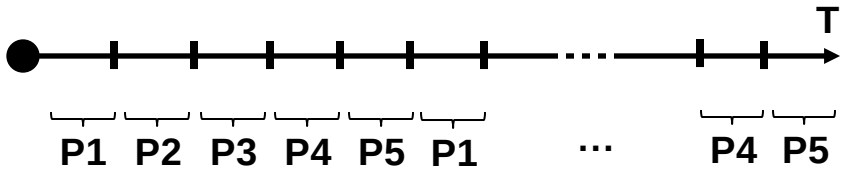
Two Scenarios (1/2)

In this section, we use Sliding Window UCB, Change Detection CUSUM UCB and EXP3 to deal with two different non-stationarity scenarios. In all cases, only one initial seed is found and activated. Furthermore, all information about the problem are assumed to be known except for the non-stationary edge activation probabilities. Such probabilities are going to be estimated using the three learners, at each round.

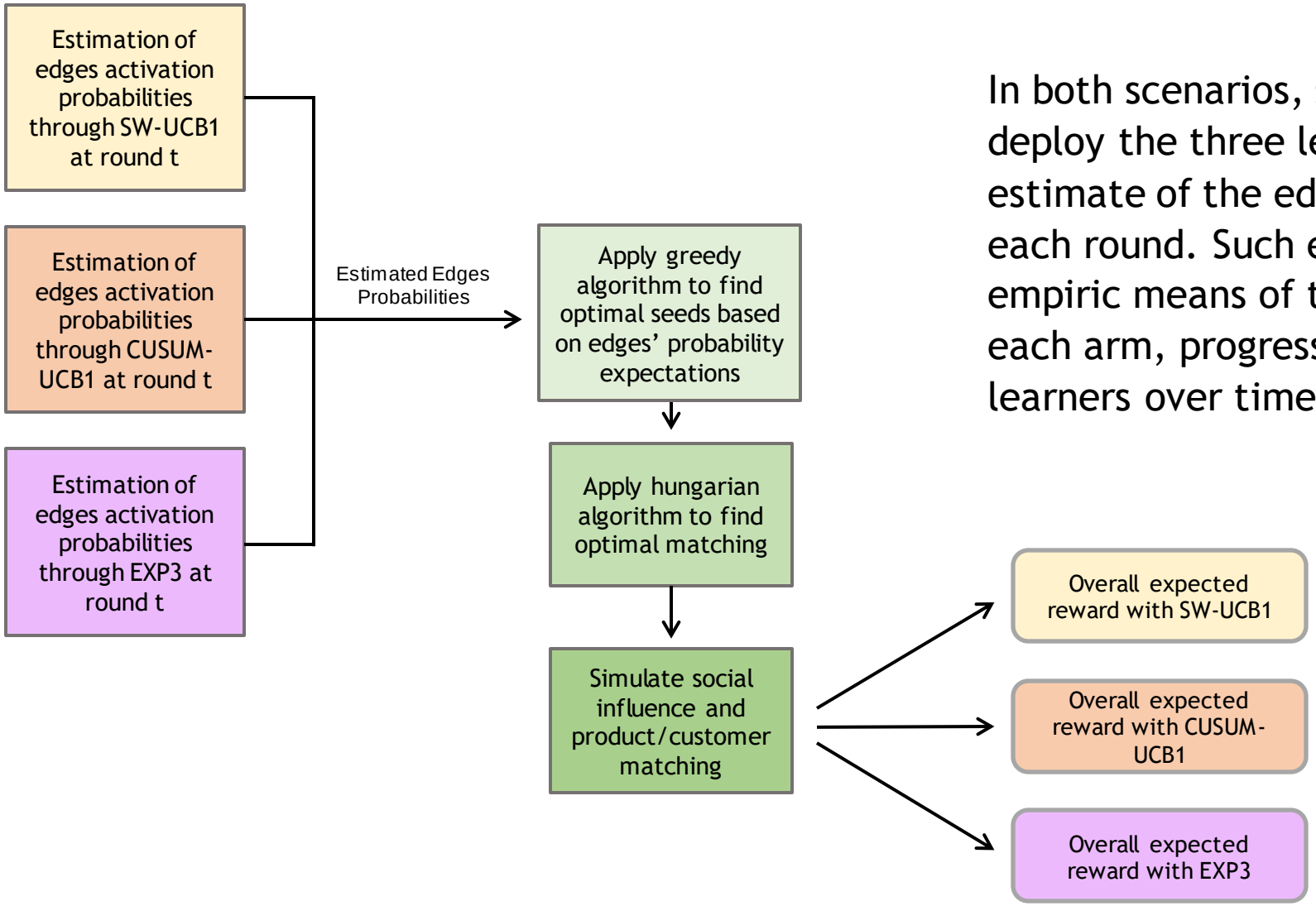
Scenario A represents a simplified version of Section 7 ("*Non-stationary environments with two abrupt changes*") in which the time horizon is divided into 3 phases with the same length.



Scenario B entails a higher non-stationarity degree in which 5 recurrent stages repeat themselves cyclically 3 times over the time horizon, for a total of 15 phases.



Two Scenarios (2/2)

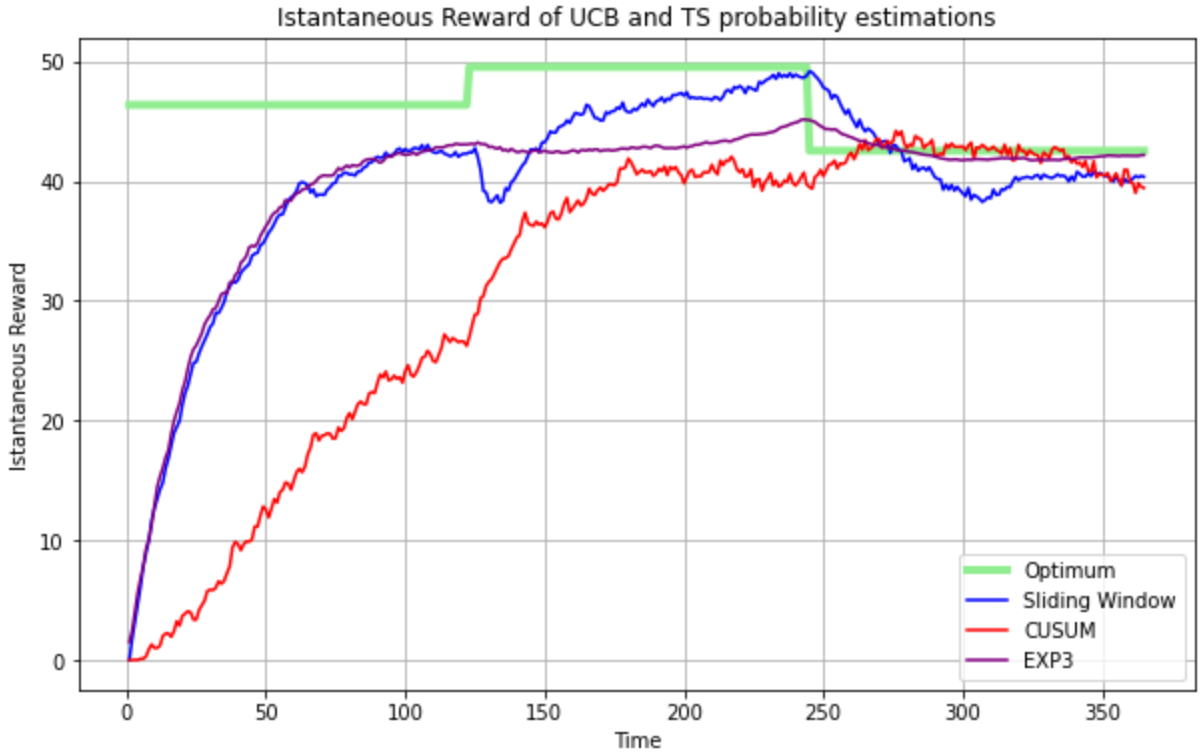


In both scenarios, similarly to Section 7, we deploy the three learners to produce an estimate of the edge activation probabilities at each round. Such estimates correspond to the empiric means of the Bernoulli rewards for each arm, progressively updated by the learners over time.

Scenario A – 3 Phases (1/4)

As shown by the picture, Sliding Window UCB appeared to be the most effective algorithm for estimating edge activation probabilities in a non-stationary setting.

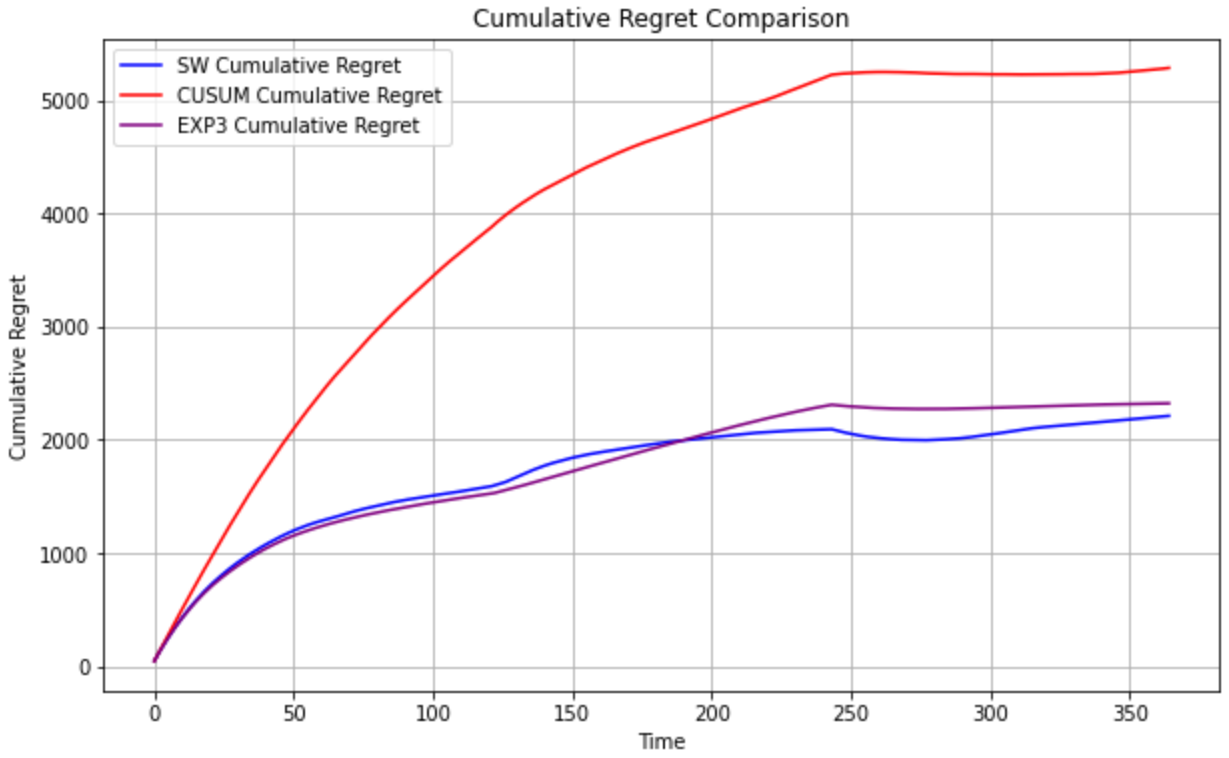
- Sliding Window UCB
  Window size = int(T/7)
- Change Detection CUSUM UCB
  M = 2, ε = 0.05, h = 1, α = 0.2
- EXP3
  γ = 0.5



Scenario A – 3 Phases (2/4)

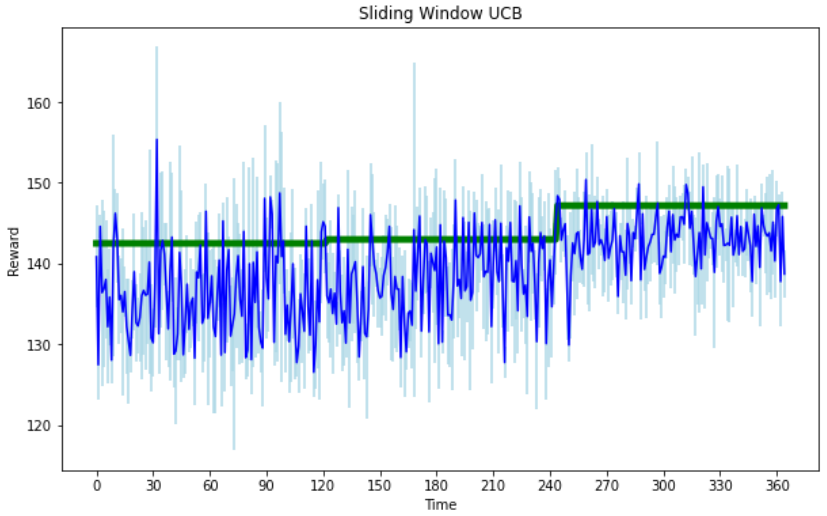
In the task of non-stationary edge probability estimation, the three algorithms have demonstrated the following cumulative regret. CUSUM UCB appears to perform much worse than the other two Learners. In all cases, the cumulative regret is sublinear over time.

- Sliding Window UCB
  Window size = int(T/7)
- Change Detection CUSUM UCB
  M = 2, ε = 0.05, h = 1, α = 0.2
- EXP3
  γ = 0.5

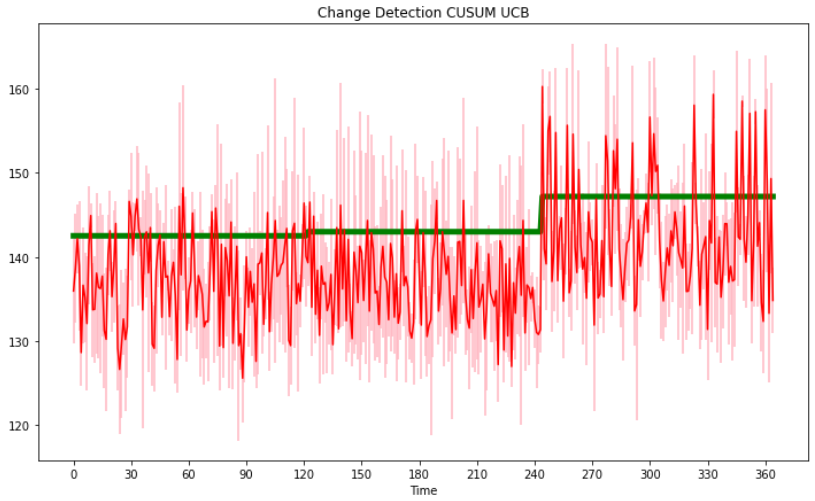


Scenario A – 3 Phases (3/4)

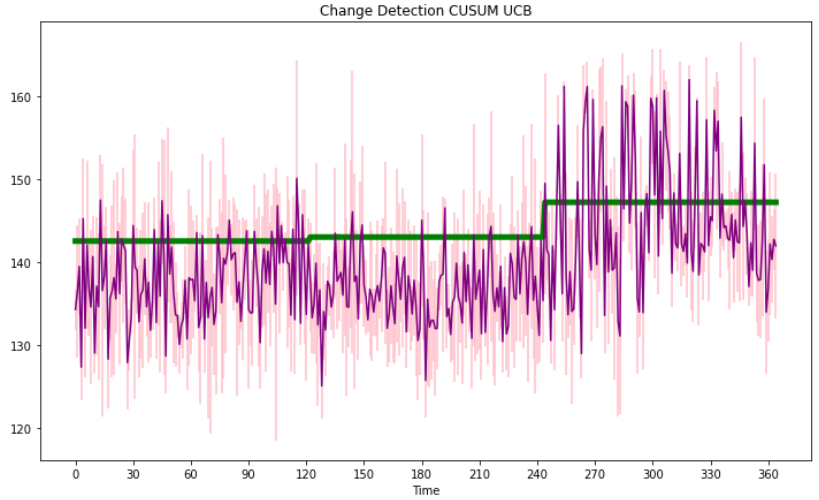
Using the estimated edge activation probabilities we simulated the optimal seed localization, the influence propagation and the matching task. Through such experiment, the following instantaneous rewards were obtained:



Sliding Window UCB



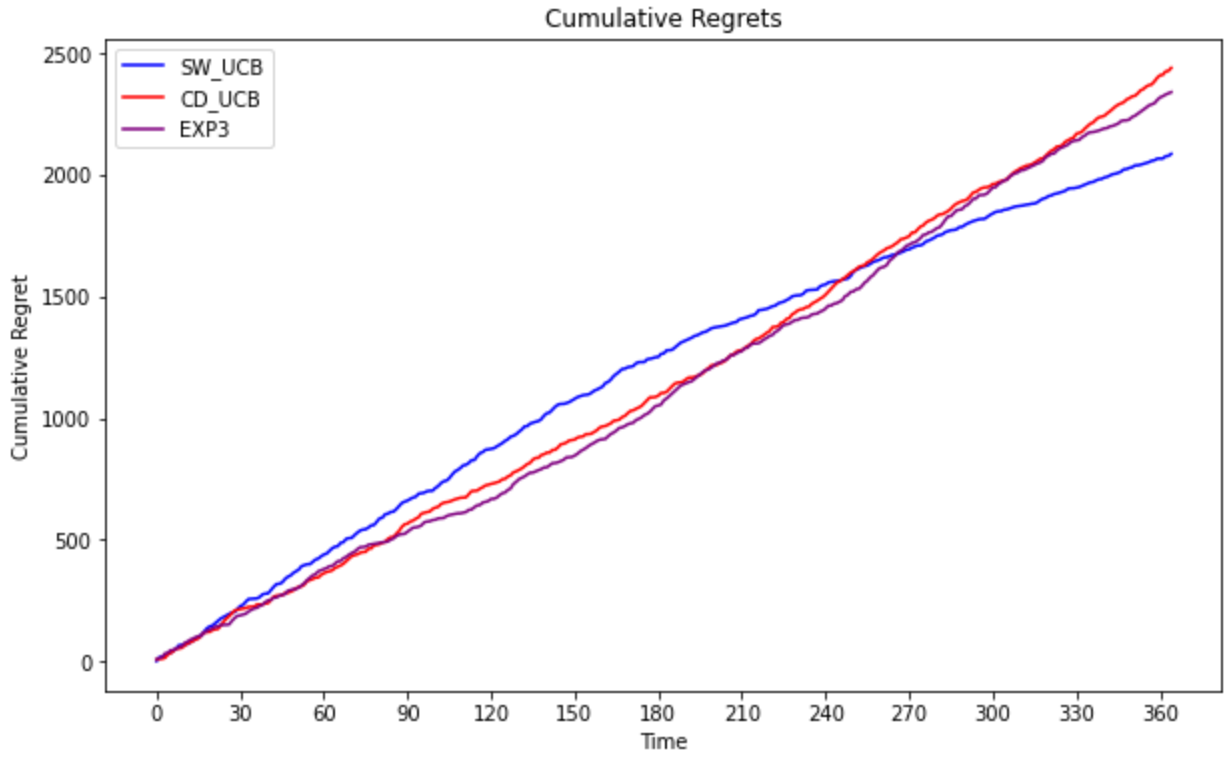
CUSUM UCB



EXP3

Scenario A – 3 Phases (4/4)

The experiment generated the following cumulative regrets.



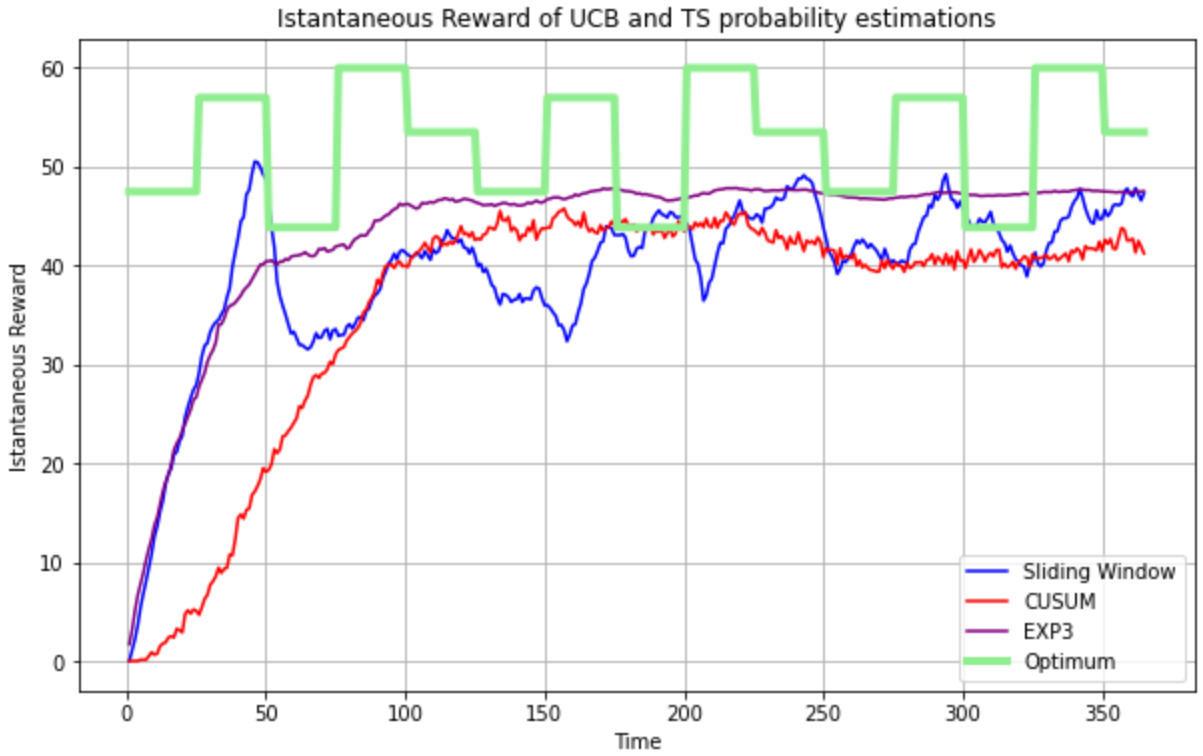
Scenario B – 15 Phases (1/4)

As displayed in the picture, on one hand, EXP3's instantaneous reward appears to converge to the optimum faster than the other two learners. On the other, Sliding Window UCB demonstrates a higher sensitivity to detecting changes over time, while yielding higher cumulative regret.

- Sliding Window UCB
  Window size = int(T/10)

- Change Detection CUSUM UCB
  $M = 2$, $\varepsilon = 0.05$, $h = 0.4$, $\alpha = 0.3$

- EXP3
  $\gamma = 0.65$
  (Considering the higher level of non-stationarity, we increased the gamma parameter to put more emphasis on the exploration effort.)



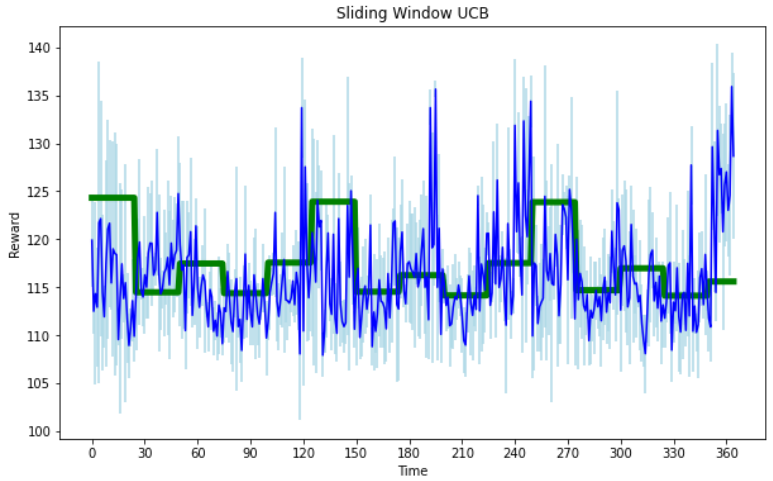
Scenario B – 15 Phases (2/4)

As suggested by the assignment, in presence of many frequent abrupt changes, EXP3 appears to outperform Sliding Window and Change Detection UCB, ensuring a lower cumulative regret in the task of probability estimation. Overall, each learner's cumulative regret increases sub-linearly over time.

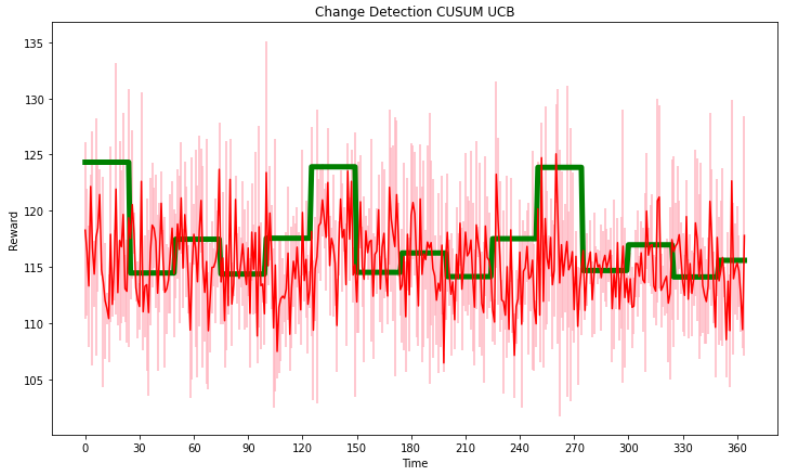
- Sliding Window UCB
  Window size = int(T/10)
- Change Detection CUSUM UCB
  $M = 2$, $\varepsilon = 0.05$, $h = 0.4$, $\alpha = 0.3$
- EXP3
  $\gamma = 0.65$
  (Considering the higher level of non-stationarity, we increased the gamma parameter to put more emphasis on the exploration effort.)

Scenario B – 15 Phases (3/4)

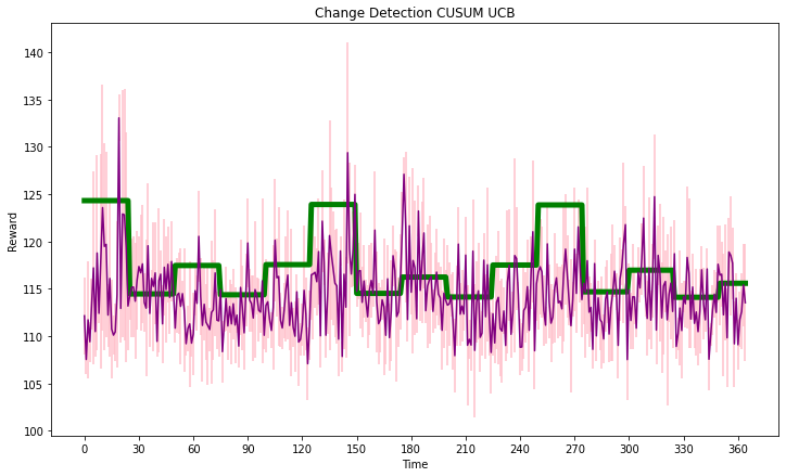
Using the estimated edge activation probabilities we simulated the optimal seed localization, the influence propagation and the matching task. Through such experiment, the following instantaneous rewards were obtained:



Sliding Window UCB



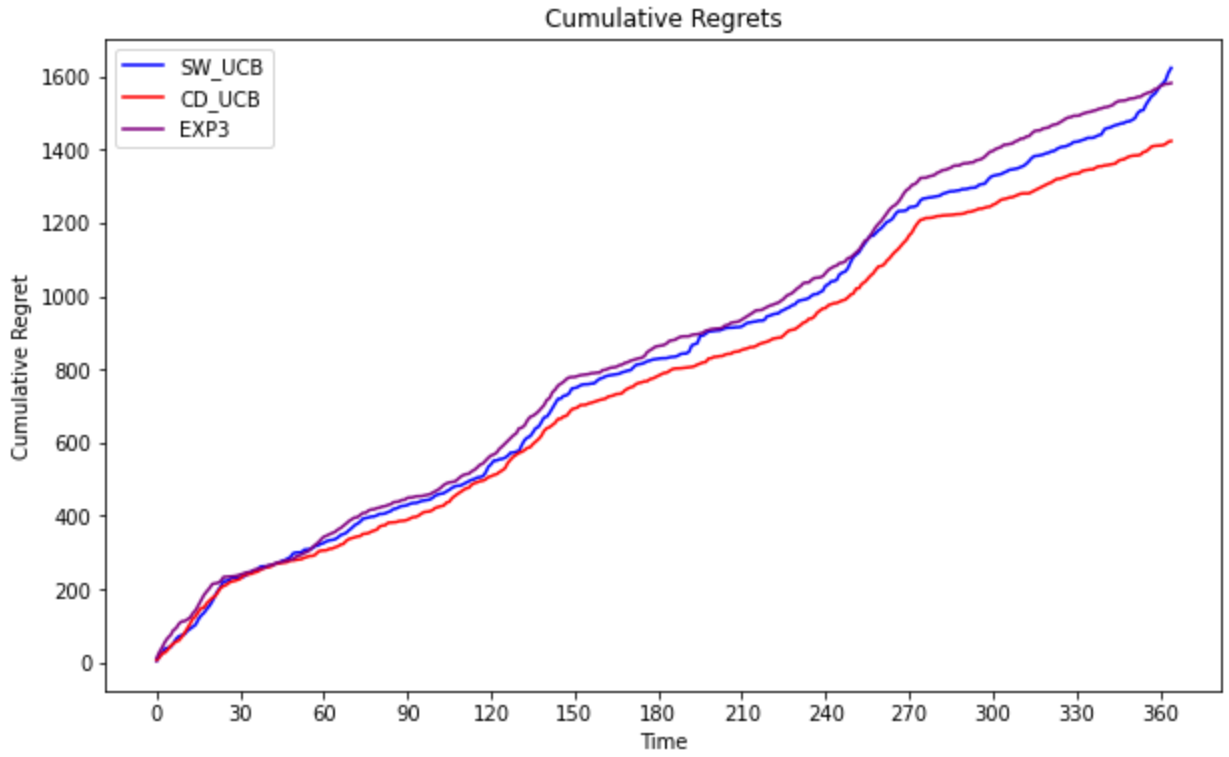
CUSUM UCB



EXP3

Scenario B – 15 Phases (4/4)

The following is the cumulative regret for the overall influence propagation and matching task.



Conclusions/Takeaways (1/1)

The results showed that the performance of the algorithms varied depending on the non-stationarity degree of the environment. ***The EXP3 algorithm performed well in the non-stationary setting with a higher non-stationarity degree***, while the non-stationary Sliding Window UCB1 performed better in the setting with less frequent changes. Lastly, Change Detection CUSUM UCB1 has generally displayed a lower performance. This may be caused by the challenges that sparsity poses on the detection of changes. In fact, due to the lack of observations for arms with activation probability equal to zero, CUSUM might take longer to detect actual changes in the probabilities, especially if the abrupt changes mostly occur in these zero-reward arms. In conclusion, the choice of algorithm depends on the specific characteristics of the environment.