Data Intelligence Application

Dynamic Pricing for a Parking System

POLITECNICO
MILANO 1863

Scotti Andrea
Paci Marco
Dello Preite Castro Gianmarco
Valladares Stefano

April 9, 2019

Contents

Abstract

This work analyses how dynamic pricing can increase the profit in the private parking industry, by applying some effective bandit algorithms. This study is carried out in a well defined setting, which is a private parking in the area of Milano. Three parkings are considered in order to define 3 different classes of users based on the location of their parking. The first location is Porta Garibaldi, one of the most famous areas of Milano; the second is Rho Fiera Milano, a neighborhood in the border of the city; and finally the third is Città Studi, a university neighborhood. This solution can be extended in a broader scenario, where drivers take advantage of a mobile application in order to find a parking lot in the city, by booking it and confirm the price per hour in advance. To simplify our model we assume a monopoly scenario where parking lots are infinite and the cost for selling a parking lot is assumed as constant and equal to zero without loss of generality. Our analysis takes also into account the fact that users in different parkings are willing to pay different prices. We model this fact by assuming a different disaggregate demand curve for each parking, and we compare the performance of the algorithm that learns the disaggregate demand curves with the one that learns the aggregate one. In general parkings are characterized by temporal phases and each of them presents a own demand curve. We identify the specific phases in our scenario and we exploit this information in order to improve the algorithms.

1 Problem contextualization

1.1 Use case example

A driver is looking for a park and decides to look for it in the private parking of Milano Porta Garibaldi. When the car is at the entrance, the server will run the algorithms to choose the price per hour for the driver. The selection of the price will take into account the context of the user, that is the chosen parking, and the phase of the year. The price is shown to them and they decide if to accept the price and park there or to refuse it and go away. Finally his decision is sent back to the server in order to update the current state of the algorithms.

1.2 Classes of users and Phases

As stated previously, the users are divided into 3 classes based on the parking lot they chose to use. This is the simplest and most effective way to identify users based on their features, as other features, like for example value of their car, age, income, etc. would be difficult to analize in our scenario. If our system were to be used online, like for example through a mobile application, this would change, but for now we decided to focus on having the customers buy the parking tickets only at the parking lot.

For each of the parking lots, we have defined a list of four phases through which the parking lots go through. The phases are all regarding the season, and are cyclical. Each phase has a different impact on each of the parkings, as they are situated in very different locations, with different demands throught the year.

- Contexts
  - Porta Garibaldi
  - Rho Fiera Milano
  - Città Studi
- Phases
  - Winter
  - Spring
  - Summer
  - Autumn

2 Algorithm Description

The algorithms used in the project have been divided into 2 main big sections, the stationary case and the non stationary case. In both cases the main goal is to find the price per hour that maximizes the profit.

The algorithms will learn the demand curve in some points, one for each arm. For the Bandits algorithm, decision of which arm to pull will be based on the expected mean provided by the arm, multiplied by the price. This is because each arm will estimate the conversion rate of the demand curve for its relative price.

2.1 Stationary case

In the stationary case we have 4 algorithms: Sequential AB testing, UCB1, Thompson Sampling and a Greedy algorithm (used as a baseline). UCB1 and Thompson sampling are bandit algorithms, while AB testing isn't (we could also say that UCB1 and TS are online algorithms, while AB testing is an offline algorithm). What this means is that the AB testing is strictly divided into phases of exploration and phases of exploitation, while the bandit algorithms mix exploration with exploitation.

These algorithms share some parameters: The time horizon, that is the number of observations of one single experiment, and the number of times the experiment is repeated. In fact, since we are simulating the environment we want to reduce the noise of the random variables that corresponds to the users in our simulation.

2.1.1 Sequential AB testing

The sequential AB testing is perhaps the easiest of the algorithms (excluding the greedy algorithm), but at the same time, as shown by the results below, the one that performs the worst. Tuning the parameters for the algorithm is extremely important to get good results. AB testing, as stated above, is divided into very clear phases of exploration, in which we continue making consistent choices, even if they are non optimal, until we have reached a certain number of samples, at which point we make the best choice and proceed to exploit it. In sequential AB testing, the exploration takes a while longer, as we need to compare more times. Once we have found the optimal arm, we enter the exploitation phase.

The algorithm consists of identifying the set of arms. From those arms, chose 2 at random. Then for a given fixed number of comparisons randomly pull 1 of the 2 arms, and save the rewards. After the fixed number of comparisons is done, compare the results of the 2 arms, and if we can say with a certain degree of confidence that one of the arms is better than the other, we discard the worse arm, put back the better arm into the poll of arms, and repeat the process until

1 arm remains, or the time horizon expires. If we can't say that one arm is better than the other based on these observations, we put both arms back in the poll and restart the process with 2 new arms. As a consequence of the behaviour of the algorithm, it is possible that by the end of the experiment, there is still more than 1 arm in the arms poll. If that were to be the case, it could mean that the number of comparisons between to arms is not high enough, or the time horizion is not big enough to compare every arm at least once (i.e 400 comparisons per pair of arms, 1600 time horizon, but 8 arms), or that the value of the accuracy is too small.

Given this behaviour there are 2 parameters that need to be set: the number of times we compare between 2 arms, and the total time horizon for the experiment.

2.1.2 Greedy

The greedy algorithm consists of just pulling the arm the looks the best, without any form of learning. It will always pull the arm with the best expected value, so we can say that this algorithm is only exploitation. In fact it perform well only when the best arm is really easy to find. We use it as a baseline for the other Bandits algorithm, since it is the simplest one. As will be shown later, it is the worst performing algorithm.

2.1.3 UCB1

The UCB1 algorithm is a Bandit algorithm, and in it's core is very simple. UCB stands for Upper Confidence Bound, and it is what defines the algorithm. The idea is to prefer exploration in case of uncertainty.

The algorithm starts by pulling each arm once. After then, it will always pull the arm with the highest upper confidence bound. In this way if there is one arm that is very good it will have an high mean, but probably also a small upper bound since the algorithm will have pulled it a lot of times. While if an arm has performed poorly but it has been pulled just a few time, it will have an high upper bound and so it will still have the possibility to be choose.

For calculating the bound we use the Hoeffding bound, that provides an upper bound on the probability that the sum of bounded independent random variables deviates from its expected value by more than a certain amount c. The upper bound we use is the following:

$$c * \sqrt{\frac{logt}{n_a(t-1)}}$$

where t is the current time, $n_a(t-1)$ is the number of times the arm has been pulled at t-1, c is the exploration-exploitation parameter. The parameter c is very

important since it defines the degree of exploration of the algorithm: the higher c is, the more exploration will have the algorithm. We experiments with different values, in particular 1, 2, $\sqrt{2}$, and the best performance was achieved with c equal to 1. This can be explain by the fact that the demand curve for the price per hour of a car is normally monotonic and it's enough easy to identify the best price. So the algorithm does not need a lot of exploration in this case.

2.1.4 Thompson Sampling

Thompsons Sampling is another bandit algorithm, that instead use a Bayesian approach to find the best price. For every arm we have a prior on it's expected value. Since the arms are distributed as Bernoulli variables the priors are distributed as Betas, starting with both alpha and beta both equal to 1; in this way we start the algorithm with a uniform distribution for each price with mean 0.5.

The algorithm then proceeds as follows: for every arm we draw a sample according to the associated Beta distribution, we then chose the arm with the best sampled value multiplied by its price, and we pull it. Finally with the obtained reward we update the values of the Beta distribution of the chosen arm according to the following formula:

$$(\alpha, \beta) := (\alpha, \beta) + (reward, 1 - reward)$$

In this case the degree of exploration is given by the prior Beta distribution of each arm. In particular the Thompson Sampling algorithm will have a good estimate of the expected mean of only the best arms with respect to the UCB1. We can notice that as we draw more and more values for one arm, the parameters of the Beta start to indicate which is the best arm in the scenario.

2.2 Non-Stationary case

In the non-stationary case we have 2 algorithms: Non stationary UCB1 and Non stationary Thompson Sampling. In this case we are assuming that during time the demand curve can change, and so the algorithm must decide the best arm to pull by taking only the last observations. To simulate it we use a Sliding Window, that will slide over the observations of each arm, and each arm will decide the best arm by taking only the observations that are inside the window. The sliding window moves at each step, and forgets the oldest value.

A very important parameter to set in the non-stationary algorithms is the length of the sliding window, as it will heavily impact the functioning of the algorithm. The selection must take into account the problem we are solved and must approximate as much as possible the true length of each phase to have a good performance on the algorithm. It must be also enough big to contain enough

observations so the algorithms can learn something, i.e. the sliding window must be set to at least the minimum time horizon that needs the stationary algorithms to perform well.

2.2.1 NS-UCB1 and NS-TS

As stated above, the algorithms perform exactly as in the non stationary case. The only big difference is the amount of data we use to calculate the Upper Confidence Bounds and the Beta distributions. That is, once a value is outside the sliding window, it has to be excluded from the calculation of the upper confidence bound and the calculation of the Beta parameters. By doing this, we are able to forget old results, which may not be providing correct information anymore.

The bound used to calculate the best arm in the UCB1 is the almost same as before, with the difference that as the current time horizon we select always the minimum between t and the windows size.

3 Contexts

The whole project is further divided into 2 other sections. These 2 sections however apply the same algorithms, aside from the AB testing, as that is a very different algorithm that operates in a very different way (fixed number of samples). This division is based on context identification. It is important to point out that context identification is not mutually exclusive with stationarity, meaning that overall our project is divided into 4 sections. Stationary and non-stationary without context identification, and stationary and non-stationary with context identification.

The context in our problem are the different parkings. In fact we notice that the area in which the parking is located affects the type of users that use it and this leads to have different demand curves, one for each area. To take into account also this we have decide to use the Contextual Bandit framework, applied in the simplest scenario where only one feature is defined, the location of the parking. For this reason we use different learners for different contexts, each one will learn the best curve for its context.

This approach is possible only because we have only one feature, but it's not scalable even when the different case for one feature grows too much. Still in our case it's the most simple and effective.

3.1 Splitting condition

Since at the beginning of the experiment we have just a few observations per context, the learners will perform poorly and will take longer to converge, so we decide to use at the beginning one more learner that learns the aggregate demand curve. In this way we start by following the aggregate demand curve, taking into account each user from each parking. When the splitting condition is met, the algorithms will follow the disaggregate curves by considering the context of the user, in our case the area of the parking from which the observation arrived. This will happen soon or later since normally the profit we can gain by disaggregating the demand curves will be higher if the features we have selected has some correlation with the price the users are willing to pay. In out case it has.

The splitting condition is based on the Hoeffding lower bound on the expected reward of the best arm. In particular the algorithm will start to follow the disaggregate curves when the following condition is true:

$$p_{c1} * LB_{c1} + p_{c2} * LB_{c2} + p_{c3} * LB_{c3} >= LB_{c0}$$

$$LB_{ci} = x_{ci} - \sqrt{\frac{\log \delta}{2 * |Z|}}$$

where p_{ci} is the probability of context i to occur, LB_{ci} is the lower bound on the best expected reward for context i, c0 is the learner that learns the aggregate demand curve, x_{ci} is the expected reward of the best arm for that context, δ is the confidence of the bound and Z the set of data. We decide to use a uniform probability since each parking we have selected can occur with the same probability and we also put $\delta = t^{-4}$ since is the most common confidence for this type of problems.

3.2 Non stationary case

In the non stationary contextual case we have that for each context the demand curve is different in different phases of the time horizon. Basically we assume that we have three contexts and each context is divided in 4 phases. The implementation is straightforward because is almost the same that the stationary case, but here the different learners for the disaggregate demand curves as well as the one that follows the aggregate demand curve will be of the non stationary type.

In this case, the algorithm at the beginning of each phase will start by following the aggregate learner, and then it will split to the disaggregates by following the same condition as the stationary case, but once for each phase. Since we don't know the true length of the phase, we decides to use the length of the window size as phase length.

4 Code Structure

The code relies on some python libraries: numpy for numerical calculations, matplotlib for plotting functions and scipy both for the calculation of the probability density function for the normal distribution (used for the AB testing) and for the calculation of the demand curves with the pchip function. Pchip is an interpolating function, it's use was very important in the project however because it is able to produce decreasing functions, which are of extreme importance for a demand curve.

The code is structured in a very straight forward way. Below are the most important parts of the code with their roles:

- **Environment class**: Simulates the environment by returning a reward every time an arm is pulled.
- **Data_Provider class**: A class containing all the information about the input data for our algorithms, meaning the values for all of the demand curves.
- **Learner class**: The super class for the learning algorithms, it stores the results returned by the environment and provides functions for updating all the values based on the observations. It is extended by all the learners, as they all add some more specific functionalities to the class.
- **Plotter class**: A class containing useful functions for drawing plots using the matplotlib library.
- **Target_Creator class**: Class that generates a curve based on the conversion rates and prices given, and the returns some values from the curvee based on the number of arms we have set. It also calculates which is the optimal arm (with it's price and conversion rate), based on the curve.
- **Sequential_AB class**: Class containing the sequential AB testing algorithm.
- **Main folder**: Folder that contains multiple main functions, that run a specific scenario of the project and displays the resulting plots. The file 'Main' will run all of the scenarios one by one (it may take some time)
- **Bandit_algs folder**: Folder that contains all the bandit algorithms. It is partitioned as follows:
  - **Bandit class**: Class that represents a single bandit algorithm.

- **Bandits class**: Class that groups bandit lgorithms based on the selected case (stat, non-stat, context or non-context).
- **Stationary folder**: Folder containing all the relevant code for the non-contextual stationary case. It includes all the relevant learners.
- **Non_Stationary folder**: Folder contaning all the relevant code for the non-contextual non-stationary case. It uncludes all the relevant learners and a non-stationary environment.
- **Contextual folder**: Folder containing the subclassing of the relevant classes that need to be modified for the contextual case (Bandit, Environment and learner) both in the stationary and non stationary case.

5 Results

5.1 Experiment setup

There are some hyperparameters that needed to be set in order to have good results from the algorithms. The first is the Time Horizon, which is the number of potential customer we have, in the given analysed time period. For the stationary case, we have decided to use a time horizon of 3000 samples, which, assuming we have around 200 potential customers a day, is equal to 15 days. It has also proven to be enough time to reach close to the optimum value.

For the non-stationary case we have set a time horizon of 6000 samples, it is longer because we are assuming that the phases are different and we need to have enough sample in each phase to make the algorithm reach the optimum. Furthermore for the non-stationary case we also need to set the length of the sliding window. We have set the sliding window to 1500 samples in our experiment. In general it must be enough longer to make the algorithm learn the curve. If, as in our case, the phases has a more or less predefined length as for the seasons, that length is a good choice for the window. These values have proven to be more than enough to have good performance with the algorithms. These values however are not 100% in line with reality, as we have stated that the real time horizon is a year. However if we were to assume to have a time horizon of a year with 200 samples a day, the execution of the algorithms would take too much time, so we have chosen to go with the time horizon of 6000 samples regardless, by assuming a smaller time horizon, but by chooding a more appropriate sliding window, the results can be generilizied easily for a more realistic time horizon.

For the contextual case, we had to set a parameter to decide when to go from using the aggregate curve to using the disaggregate curves. The condition used is based on the Hoeffding lower bound, and it checks whether the mean of the mean of the best arms for each of the disaggregate curves is higher than the mean of the best arm of the aggregate curve. Note that the mean for the disaggregate curves could have been a weighted average if the probability for each context was different. Refer to section 3.1 for a more detailed explanation.

Another important parameter is the number of arms we use. The more arms we have the more accurate will be the optimum value, but the algorithms have a temporal complexity that increases linearly with the number of arms, so we need to take it into account. At the end we decide to use 8 arms for every algorithm. In fact, with less arms sometimes we don't have arms near to the optimum of the curves, while with more arms the algorithm doesn't improve so much and the running time is slightly higher.

Finally, we had to set the number of times each experiment would be executed. This is of crucial importance in decreasing the noise of the results, but at the same

time, it significantly increases the computational cost of the execution. After having experimented some values, we verify that any value below 50 still had some sort of noise, and any value above 300 increased the running time of the algorithm too much. We therefore decided to set it at 100.

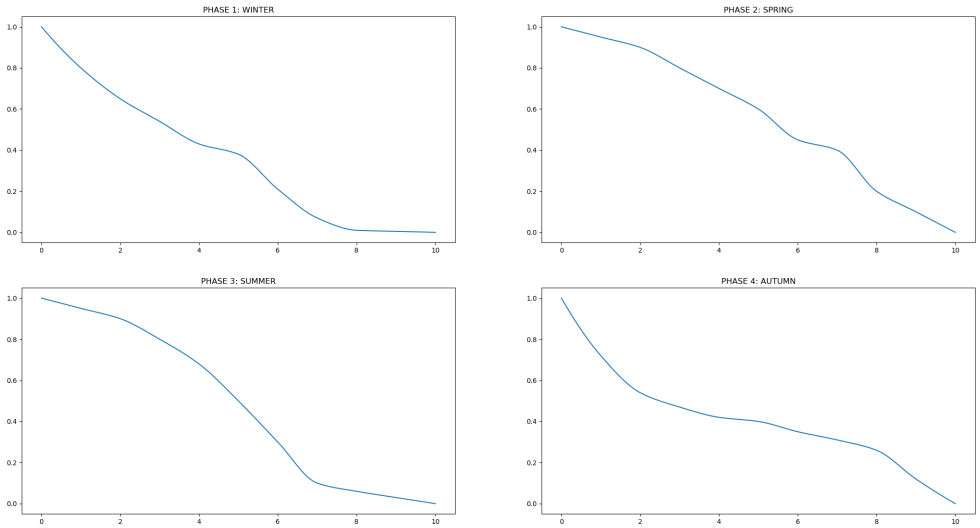
We have chosen, as stated before, three different parking lots to run the experiment on. Furthermore we divided the time period into four different phases, which we chose to be the different seasons of the year, as they have a big impact on the number on people going to the different parkings. Below we show the conversion rates used.

```
# Context 1 Rho Fiera
dis_cnv_rates[0][0] = np.array([1, .80, .65, .54, .43, .38, .21, .07, .01, .005, 0])  # Winter
dis_cnv_rates[0][1] = np.array([1, .95, .90, .80, .70, .60, .45, .40, .20, .10, 0])   # Spring
dis_cnv_rates[0][2] = np.array([1, .95, .90, .80, .68, .50, .30, .10, .06, .03, 0])   # Summer
dis_cnv_rates[0][3] = np.array([1, .72, .54, .47, .42, .40, .35, .31, .26, .12, 0])   # Autumn

# Context 2 Porta Garibaldi
dis_cnv_rates[1][0] = np.array([1, .86, .78, .75, .64, .53, .32, .20, .10, .05, 0])   # Winter
dis_cnv_rates[1][1] = np.array([1, .95, .90, .76, .61, .52, .45, .30, .15, .10, 0])   # Spring
dis_cnv_rates[1][2] = np.array([1, .99, .92, .86, .72, .66, .54, .42, .20, .10, 0])   # Summer
dis_cnv_rates[1][3] = np.array([1, .60, .54, .46, .40, .32, .31, .24, .12, .05, 0])   # Autumn

# Context 3 Città Studi
dis_cnv_rates[2][0] = np.array([1, .80, .70, .60, .50, .40, .30, .20, .10, .005, 0])  # Winter
dis_cnv_rates[2][1] = np.array([1, .85, .80, .66, .40, .37, .31, .26, .12, .08, 0])   # Spring
dis_cnv_rates[2][2] = np.array([1, .50, .35, .22, .15, .07, .02, .009, .005, .001, 0])# Summer
dis_cnv_rates[2][3] = np.array([1, .85, .80, .66, .40, .37, .31, .26, .12, .08, 0])   # Autumn
```

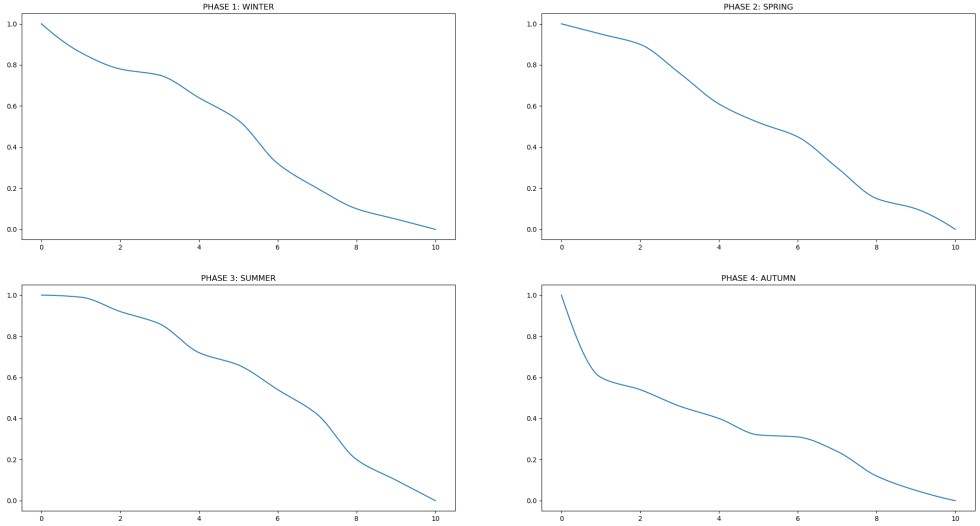
The demand curves used have been chosen at random, but considering that they should be monotonically decreasing, as in the real case. In general, winter is not a good time for parkings in Milan, as most people prefer to stay home due to the cold. In fact, in this season people are less willing to pay more for a parking. Spring is a great season all around, it brings some big events in Rho Fiera, Porta Garibaldi is crowded most of the time, and Città Studi is always full of students preparing exams. Summer is somewhat different. It's the season when most people decide to actually leave Milan and go enjoy the warm weather somewhere else. Some events still happen in Rho Fiera, and Porta Garibaldi is crowded with tourists and people that want to enjoy their time in Milan. Città Studi however is empty because the university is closed. Autumn has less people going out as the weather starts getting cold again. Not many events in Rho Fiera and Porta Garibaldi starts being less appealing. As for Città Studi, autumn is the same as Spring.

5.2 Demand curves

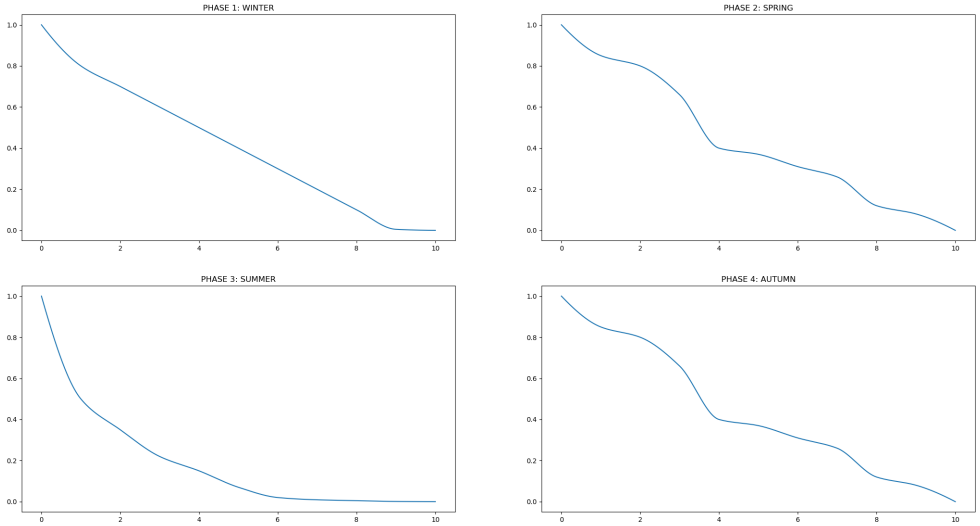
5.2.1 Rho Fiera



5.2.2 Porta Garibaldi



5.2.3 Città Studi



5.3 Results

In this section we are going to plot the results we have obtained from running our algorithms in all the possible cases. In particular we report the reward and the regret obtained by each of them. These are two very well-knowed metrics for comparing Bandits algorithm, since they highlight the time for the algorithm to get the optimum, as well as the amount of regret we got before the algorithm finds the optimum price. For the contextual case we report only the regret curves, since we were not able to run the algorithms for enough experiments to have a plot of the reward without noise. This cases are computationally inefficient and for a more realistic results they should be run on a more powerfull machine.

5.3.1 Stationary

In the stationary case, results are as expected. AB testing performs the worst of all the learning algorithms. While Thompson sampling performs considerably better than UCB1, and converges a lot faster. For this case we also show the reward and demand curve and the mean value of each arm. As we expect they are very accurate for the best arms, while less accurate for the worste ones.

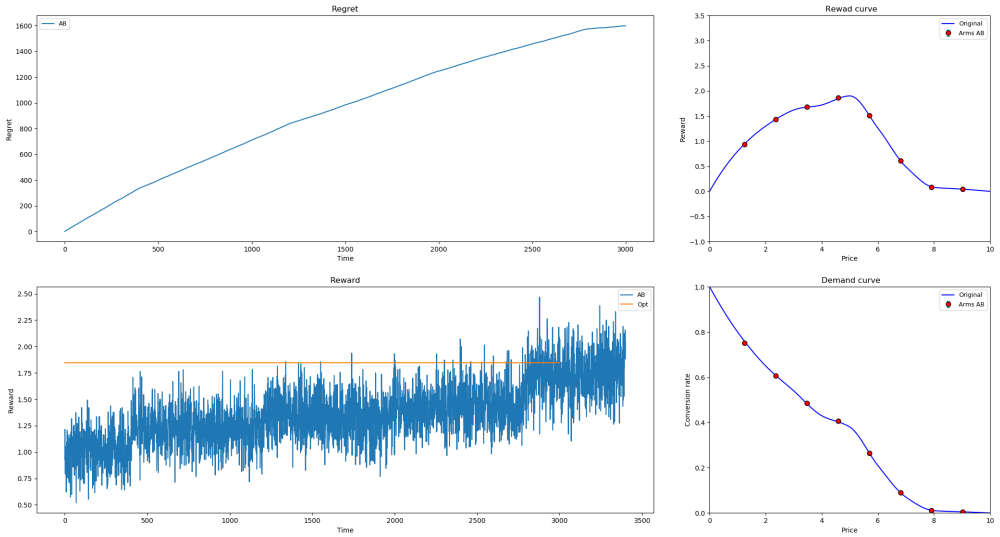


Figure 1: AB Testing

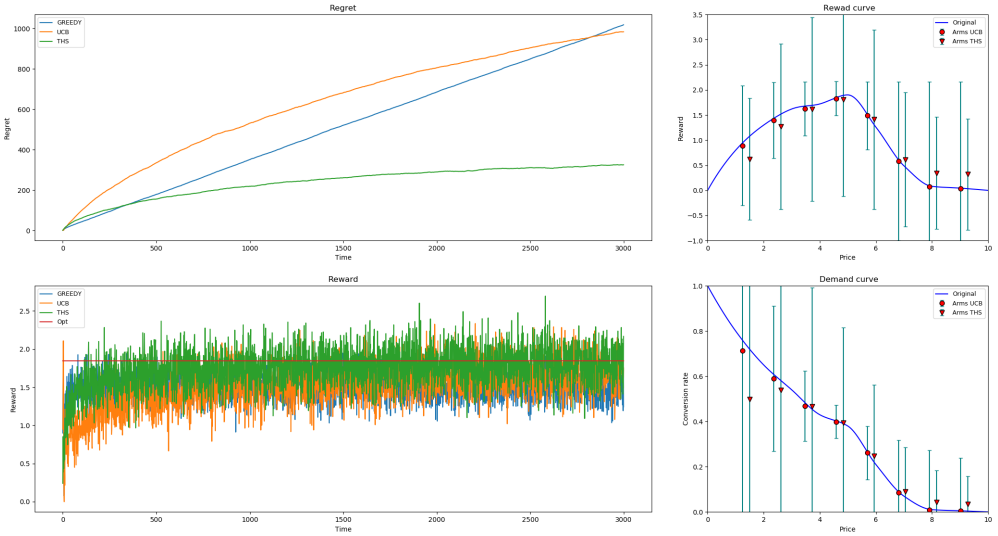


Figure 2: Bandit stationary case

5.3.2 Non stationary

In the non-stationary case, we can see that in the first phase the non-stationary algorithm performs as the stationary. From the second phase on, the non-stationary algorithms don't suffer a big regret since they forget old samples. Even in this case the Thompson Sampling outperforms the UCB1, as we expect.

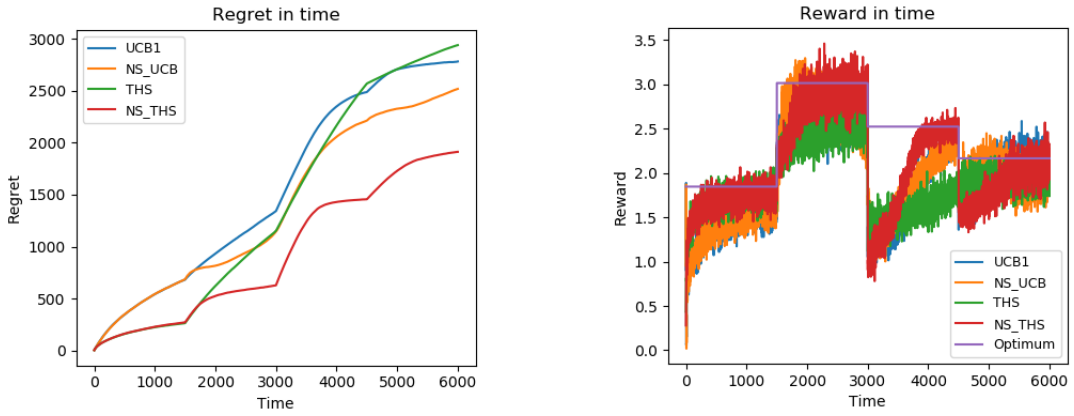


Figure 3: Bandit non-stationary case

5.3.3 Contextual stationary

In the contextual case we have some interesting unexpected results. The aggregate and disaggregate algorithms outperforms the contextual algorithm for the UCB algorithm, while in the Thompson Sampling case the disaggregate one is almost the same as the contextual. The curves selected for this test are those of the second phase for each context and they are very similar. In fact we should select curves more idfferent to have better result, but we want our experiment to be as close as possible to the reality and so we chose a phase at random. For the other phases the results are also similar.

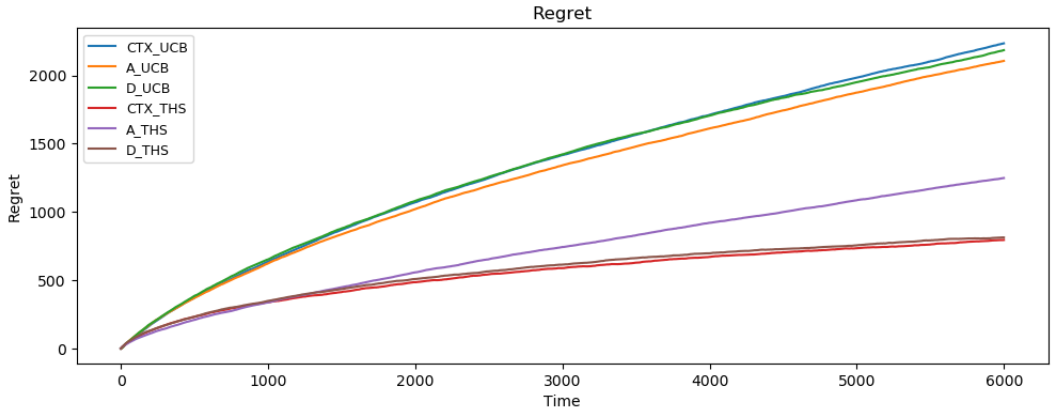


Figure 4: Contextual bandit stationary case

5.3.4 Contextual non stationary

The contextual non stationary case is the most complex of them all, but the result are consinstent with what we expected this time. The algorithms seem to not have been able to reach the convergence within the time horizon, hence the regret is still increasing. A solution to this would to simply have more samples within each of the phases, in order to reach the optimal value before the phase switches. Once again we see that Thompson Sampling outperfoms UCB1. The result also shows that the non contextual one at the end outperforms the aggregate and disaggregate algorithms, this is because at the beginning of each phase it starts by considering every sample, but after some time steps it uses the contextual information to decide the price, obtaining at the end of the phase an higher cumulative reward with respect the others two.

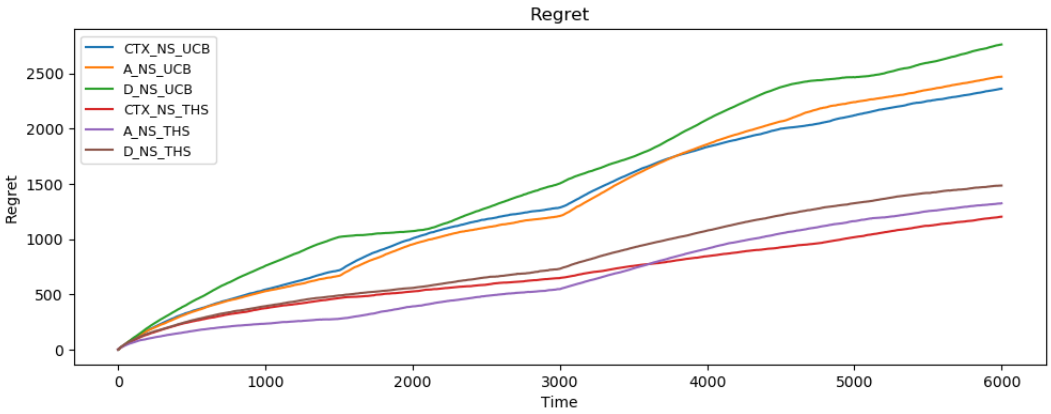


Figure 5: Contextual Bandit non-stationary case

6 Conclusions

This research project has the objective to analyze how various algorithms perform in the decision of the pricing per hour in a parking in Milan applied to the Bandit settings. We have considered different scenarios, by taking into account different context for which the price could differ, and also different phases during the year in which the demand curves are different.

In conclusion we can say that there is much to be gained from using Bandit algorithms to solve the dynamic pricing problem. Even in a scenario in which one may not think to apply such 'complex' algorithms, such as for a parking lots, we can see huge decreases in regret (or profit) from the performance of such algorithms in our simulated environment.

We are satisfied about the results, since they show in almost every case a good behaviour even if our problem has been never applied in the Bandit framework. The most critical one is the contextual case, in which we show that the contextual algorithm is not capable to beat the others. We suggest to increase the number of context, by considering more parkings in different areas and increase the number of sample. Another idea could be increase the number of features of each context. In fact, in our settings we only have considered one feature, that is the location of the parkings. We can also say, that the optimal algorithm to choose is the Thompson Sampling, as it has outperformed all the others in every single scenario.

We can conclude that an increase in the number of samples, even at the increased cost of computational power, may prove to be useful in order to further decrease the regret, given that in our problem the data are easy to collect. We also suggest that for a true test this algorithm should be tried for a period in a real parking, or at least they should be tested with real data.