Advertising and Pricing

Gabriele Daglio, Federico Di Cesare, Jacopo Germano

July 11, 2021

1 The setting

Advertising is used to attract users on an ecommerce website that sells only one type of item. Each day, a stochastic number of auctions A is run by the ad publisher, each auction corresponding to a different user. The users are characterized by two binary features F_1 and F_2, which are independent and described by their probabilies θ_1 and θ_2 of assuming the value $True$ for a specific user. Each user belongs to one of three classes, and the classes are determined by the combination of the two features. For each class $c \in C$ the following functions are modeled:

- A stochastic number of daily clicks of new users (i.e., that have never clicked before on the ads), represented by the discrete random variable $N_{c,b}$ such that $\mathbb{E}[N_{c,b}] = n(c,b)$, where $b \in B$ is a bid value.

- A conversion rate function providing the probability that a user will buy the item at a certain price, $r(c,p)$, where $p \in P$ is the price. For each user of class c that has clicked on the ad, a Bernoulli random variable $D_{c,p} \sim Bern(r(c,p))$ indicates whether the user bought the product ($D_{c,p} = 1$) or not ($D_{c,p} = 0$), such that $\mathbb{E}[D_{c,p}] = r(c,p)$. We call this distribution $ClickConverted(c,p)$ and therefore $D_{c,p} \sim ClickConverted(c,p)$.

- A probability distribution $FutureVisits(c)$ over the number of times a user of class c will come back to the ecommerce website to buy that item by 30 days after the first purchase. In other words, when a user makes a purchase, they are somehow likely to make more purchases in the near future, and after that they will leave the website forever. For each user of class c, a discrete random variable $F_c \sim FutureVisits(c)$ indicates the number of times that the user came back, and for each class c the function $f(c)$ is defined such that $\mathbb{E}[F_c] = f(c)$

- A probability distribution $CostPerClick(c,b)$. For each click, the random variable $C_{c,b} \sim CostPerClick(c,b)$ represents the amount that is paid to the ad publisher, such that $\mathbb{E}[C_{c,b}] = k(c,b)$ and $\mathbb{P}(C_{c,b} \leq b) = 1$.

A margin function $m(p)$, where $p \in P$ is a price, is available to indicate how much profit is obtained if the an item is sold at the price p.

1.1 The binary features

The features F_1 and F_2 are independent and are governed by the parameters θ_1 and θ_2. At the generation of the environment, the parameters θ_1 and θ_2 are randomly sampled, which represent the distribution of the feature values being for each user u: $\Pr(F_{1,u} = True) = \theta_1$ and $\Pr(F_{2,u} = True) = \theta_2$. Since the features are independent, for each combination we can compute the likelihood as follows:

$$\begin{aligned}
\tilde{l}_{TT} &= \theta_1\theta_2 \\
\tilde{l}_{TF} &= \theta_1(1-\theta_2) \\
\tilde{l}_{FT} &= (1-\theta_1)\theta_2 \\
\tilde{l}_{FF} &= (1-\theta_1)(1-\theta_2)
\end{aligned}$$

1.2 The number of new daily clicks $N_{c,b}$

First, the number of auctions A run by the ad publisher is determined. The random variable A is distributed as a Poisson $A \sim Poisson(\lambda_a)$, with the mean λ_a being randomly chosen when the environment is generated.

The number $N_{c,b}$ of new daily clicks of users belonging to class c is determined as follows: given the number of auctions A, the likelihood of each combination of the two binary features is used to sample from a Multinomial distribution:

$$(A_{TT}, A_{TF}, A_{FT}, A_{FF}) \sim Multinomial(A, (\tilde{l}_{TT}, \tilde{l}_{TF}, \tilde{l}_{FT}, \tilde{l}_{FF}))$$

Where A is the number of tries and $(\tilde{l}_{TT}, \tilde{l}_{TF}, \tilde{l}_{FT}, \tilde{l}_{FF})$ is the vector of probabilities. This process basically assigns a combination of the features to each user involved in an auction of the ad publisher, according to the likelihood of each feature. The result, $(A_{TT}, A_{TF}, A_{FT}, A_{FF})$, is the number of auctions run by the ad publisher for users of each combination.
For each $comb \in \{TT, TF, FT, FF\}$ it holds $\mathbb{E}[A_{comb}] = \lambda_a \tilde{l}_{comb}$ (Appendix A)

For each auction, a Bernoulli random variable is sampled to determine if the owner of the e-commerce has won the auction. We adopt the following assumption about the probability of winning an auction:

Assumption (Agnostic Publisher)**.** *The probability of winning an auction does not depend on the features that characterize the user, but only depends on the bid value b.*

The meaning of the Agnostic Publisher assumption is that a change in the bid will change the number of users seeing the ad but will not change the percentage of users for each class. In other words, an increase in the auctions won will reflect in an increase in the number of clicks with the same proportion on all the classes.

We therefore define the function $v(b)$ representing the probability of winning one auction: it needs to be a monotonically increasing function of the bid ranging from 0 to 1. We chose a sigmoidal function for this purpose:

$$v(b) = \frac{1}{1 + e^{-\bar{z}(b-\bar{b})}}$$

where $\bar{z} > 0$ and $\bar{b} > 0$ are randomly sampled when the environment is generated.

For simplicity we will assume that all the users that are displayed the ad will also click on it, therefore in our model the number of auctions won and the number of daily clicks coincide.

Finally, the number of clicks of users described by the combination of features *comb*, which is equivalent to the number of auctions won, is sampled from a Binomial:

$$\tilde{N}_{comb,b} \sim Binomial(A_{comb}, v(b))$$

where A_{comb} is the number of tries and $v(b)$ the probability of success of one try, and assuming that the function $combs(c)$ maps each class c to the set of combinations of features that are covered by that class, we can compute

$$N_{c,b} = \sum_{comb \in combs(c)} \tilde{N}_{comb,b}$$

Defining the likelihood l_c of class c as

$$l_c = \sum_{comb \in combs(c)} \tilde{l}_{comb}$$

we obtain the expression of the expected value of $N_{c,b}$

$$n(c,b) = \mathbb{E}[N_{c,b}] = \lambda_a l_c v(b)$$

(Appendix B)

1.3 The conversion rate $r(c,p)$

For each class $c \in C$, the function $r_c(p)$ used to model the conversion rate of users belonging to that class must have the following properties:

- $r_c(0) \approx 1$: The user will be very likely to buy the product if it comes for free.
- $\lim_{p \to +\infty} r_c(p) = 0$: As the price goes to infinity, the probability that the user will buy it goes to zero.

- $r_c(p)$ is monotonically decreasing with respect to p: an increase of the price will never increase the probability that the user will buy it.

The function $r(c,p)$ is then defined as: $r(c,p) := r_c(p)$. Despite the fact that the functions $r_c(p)$ could be in principle defined in completely different ways, in our implementation we chose to use only (reflected, translated and horizontally scaled) sigmoidal functions:

$$r_c(p) = \frac{1}{1 + e^{-z_c(P_c - p)}}$$

Where P_c, the inflection point of the sigmoid, can be seen as the average reserve price of the users of the class and z_c can be seen as the concentration of the reserve prices of the users around the average: if z_c is small the reserve prices of the many users will be more distributed across the domain and the function will be more flat, while as z_c grows the reserve prices of the many users will be more concentrated around the average and at the point P_c there will be a rapid transition from "buy" to "don't buy".

1.4 The future visits F_c

We modeled the future visits with a Poisson random variable, such that $F_c \sim Poisson(f(c))$ and the mean $f(c)$ is constant and randomly sampled for each class when the environment is generated. We call the resulting distribution $FutureVisits(c)$, therefore $F_c \sim FutureVisits(c)$.

1.5 The cost per click $C_{c,b}$

The price paid for each click is in principle a stochastic function of the bid and of the user class. To limit the complexity of the model, we chose to model it such that the mean $k(c,b)$ is a percentage of the bid and the variable is always equal to its mean.

$$k(c,b) = u_c\ b$$
$$\mathbb{P}(C_{c,b} = k(c,b)) = 1$$

With the percentage $0 < u_c < 1$ being randomly sampled for each class when the environment is generated. We call this distribution $CostPerClick(c,b)$, therefore $C_{c,b} \sim CostPerClick(c,b)$.

1.6 Margin function $m(p)$

At the generation of the environment, the base price p_{base} of the item is randomly sampled. This is the price that the seller has paid to produce the item, therefore the margin function $m(p)$ is defined as follows:

$$m(p) = p - p_{base}$$

And it represents the profit that the seller makes by selling one item at price p.

2 Step 1

The goal is to maximize the expected profit over a single day, where the future visits of a user are considered to contribute in expected value to the profit of the day of the first visit.

For each class c, we consider:

- The random variable $N_{c,b}$ representing the number of new clicks of users of class c.
- The sequence $(C_{c,b,i})_{i=1,\dots,N_{c,b}}$ of random variables representing the cost paid for each click i, such that $C_{c,b,i} \sim CostPerClick(c,b)$
- The sequence $(D_{c,p,i})_{i=1,\dots,N_{c,b}}$ of random variables representing whether user i of class c purchased the item, such that $D_{c,p,i} \sim ClickConverted(c,p)$
- The sequence $(F_{c,i})_{i=1,\dots,N_{c,b}}$ of random variables representing the number of future visits of the user i of class c, such that $F_{c,i} \sim FutureVisits(c)$

With these variables we can express the expected profit as follows:

$$ExpectedProfit(p,b) = \mathbb{E}\left[\sum_{c\in C}\sum_{i=1}^{N_{c,b}}\Big(D_{c,p,i}(1+F_{c,i})m(p) - C_{c,b,i}\Big)\right]$$

And therefore formulate the optimization problem as follows:

$$\arg\max_{p,b}\ ExpectedProfit(p,b)$$

With some manipulations using the properties of the expected value (Appendix C), the expected profit can be expressed as follows:

$$ExpectedProfit(p,b) = \sum_{c\in C} n(c,b)\Big(m(p)r(c,p)(1+f(c)) - k(c,b)\Big)$$

Obtaining the following formulation of the optimization problem which depends only on the means of the distributions:

$$\arg\max_{p,b}\ \sum_{c\in C} n(c,b)\Big(m(p)r(c,p)(1+f(c)) - k(c,b)\Big)$$

Under the Agnostic Publisher assumption, the following interesting result holds:

Lemma (Bid Independent Price Hierarchy). *If $ExpectedProfit(p_1,\bar{b}) \geq ExpectedProfit(p_2,\bar{b})$ for some bid value $\bar{b}$, then $ExpectedProfit(p_1,b') \geq ExpectedProfit(p_2,b')$ for every possible bid value b'.*

Proof. see Appendix D. ∎

As a consequence of this result, we developed an algorithm that:

- Finds the optimal price $p^* \in P$ that maximizes $ExpectedProfit(p, \bar{b})$ for a fixed bid value $\bar{b}$ (We take the median of the set of possible values). The time complexity of this step is $O(|P|)$

- Finds the optimal bid value $b^* \in B$ that maximizes $ExpectedProfit(p^*, b)$, using the optimal price p^* found at step 1, which is still optimal thanks to the above lemma. The time complexity of this step is $O(|B|)$

- The solution is (p^*, b^*).

And computes the optimal solution with a time complexity of $O(|P| + |B|)$.

3 Step 2

We can model the online version of the above optimization problem as follows: each day corresponds to a round, and the learning horizon is $H = 365$ rounds. Before each round the learner specifies the price p and the bid b that will be used, while at the end of the round j the learner will receive the following information:

- The number of auctions a_j that were run by the ad publisher during that round
- The number of new clicks n_j received during that round
- The number of purchases of new users s_j that happened during that round
- The total cost per click that was paid to the ad publisher c_j
- The total number f_{j-30} of subsequent purchases done by users that did the first purchase on round $j - 30$. (If $j < 30$, this information will be omitted).

It must be noticed that there is a partially delayed feedback: in fact, the future visits of the users are defined as the number of subsequent purchases in the next 30 days and the learner will need to wait as many rounds to know the number of future visits that was realized, while they will be immediately aware of the realization of the other values.

The learner's goal should be to estimate the expected profit of all the pricing/bidding strategies (p, b), in order to employ the most profitable one. At round $i > 30 + |P|$, this estimation can be obtained by computing the expected value according to the formula defined in step 1, estimating:

- The average number of daily auctions $\overline{a}$ as the sample mean of the previously observed $a_j s$:

$$\overline{a}_i = \frac{1}{i} \sum_{j=0}^{i-1} a_j$$

- For each bid value b, the average number of new clicks $\overline{n_b}$ as the mean of the observed n_j, considering only the rounds where the bid b was employed:

$$\overline{n_b}_i = \frac{1}{|\{j : b_j = b\}|} \sum_{j \in \{j : b_j = b\}} n_j$$

- For each price p, the conversion rate $\overline{r_p}$ as the ratio between the number of purchases and that of new clicks, considering only the rounds where the price p was employed:

$$\overline{r_p}_i = \frac{\sum_{j \in \{j : p_j = p\}} s_j}{\sum_{j \in \{j : p_j = p\}} n_j}$$

- For each bid value b, the average cost per click $\overline{c_b}$ that was paid to the ad publisher c_j as the ratio between the total cost paid and the total number of clicks, considering only the rounds where the bid b was employed:

$$\overline{c_b}_i = \frac{\sum_{j\in\{j:b_j=b\}} c_j}{\sum_{j\in\{j:b_j=b\}} n_j}$$

- For each price p, the average number $\overline{f_p}$ of subsequent purchases done by users as the ratio between the total number of subsequent purchases of users of round j and the number of purchases of round j, considering only the rounds where the price p was employed and for which the delayed feedback has been received:

$$\overline{f_p}_i = \frac{\sum_{j\in\{j:p_j=p,j\le i-30\}} f_j}{\sum_{j\in\{j:p_j=p,j\le i-30\}} s_j}$$

- For each bid value b, the probability $\overline{w_b}$ of winning the auction as the ratio between the number of new clicks and the number of auctions (recall that we are assuming that every user which is displayed the ad will also click on it), considering only the rounds where the bid b was employed:

$$\overline{w_b}_i = \frac{\sum_{j\in\{j:b_j=b\}} n_j}{\sum_{j\in\{j:b_j=b\}} a_j}$$

After all these estimates have been computed, the learner can make a projecton of the expected profit $\widehat{exp(p,b)}_i$ of the strategy (p, b) as:

$$\widehat{exp(p,b)}_i = \overline{a}_i\overline{w_b}_i\left(m(p)\overline{r_p}_i(1+\overline{f_p}_i) - \overline{c_b}\right)$$

The estimation can be done efficiently because none of the quantities to estimate depends both on p and b. It may be non trivial to notice that the estimate of $\overline{f_p}$ can be computed summing the sample that come from rounds where different bids were used, we show this in Appendix E. Using the estimation the learner will choose (p_i, b_i) and the learning process will move one round forward. Since the learner is optimizing on p and b, they should not use the expected profit but rather an optimistic estimate of the expected profit, which can be obtained by applying to the conversion rate $\overline{r_p}_i$ and to the winning probability $\overline{w_b}_i$ optimistic exploration techniques such as upper confidence bounds or Thompson sampling and using such optimistic values instead of the sample means.

Note: *The number of rounds* $30 + P$ *after which the learner begins to compute the estimates is to ensure that all the prices have received at least one delayed feedback in the case the learner performs round robins for the first* P *rounds*

4 Step 3

In order to learn in an online fashion the best pricing strategy for a fixed bid value, we have implemented a python object called *OptimalPriceLearner* and one called *PriceBanditEnvironment*, which has internally an instance of the *Environment* object. Upon the creation of an instance of a *PriceBanditEnvironment*, the underlying *Environment*, the set of possible prices and the fixed bid value must be supplied as arguments. The bandit environment hides the actual prices P from the learner, and instead it shows a bandit-like set of arms numbered from 0 to $|P| - 1$.

The *PriceBanditEnvironment* exposes the method *pull_arm_not_discriminating(arm: int)*, which returns the aggregated data:
new_clicks, purchases, tot_cost_per_clicks, (past_arm, past_future_visits).

These data implement the round output specified at step 2, with the exception of *past_arm* which is just a reminder of the arm that was chosen at the round in which the delayed feedback started.
The learning loop is as follows:

```
def learn(self, n_rounds: int):
    self.round_robin()

    while self.current_round < n_rounds:
        self.learn_one_round()

def learn_one_round(self):
    arm = self.choose_next_arm()
    self.pull_from_env(arm=arm)

def choose_next_arm(self):
    return int(np.argmax(self.compute_projected_profits()))
```

It is clear that the learner chooses to pull the arm that has the highest projected profit, which is, for each arm, the profit computed with an optimistic estimate of the conversion rate associated to the arm itself.
Let's see how the learner computes the projected profits:

```
def compute_projected_profits(self):
    new_clicks = self.compute_new_clicks()
    margin = np.array([self.env.margin(a) for a in range(self.n_arms)])
    crs = self.compute_projection_conversion_rates()
    future_visits = self.compute_future_visits_per_arm()
    cost_per_click = self.tot_cost_per_click / \
                     sum_ragged_matrix(self.new_clicks_per_arm)
```

```python
        projected_profit = simple_class_profit(
            margin=margin, conversion_rate=crs, new_clicks=new_clicks,
            future_visits=future_visits, cost_per_click=cost_per_click
        )

        return projected_profit
```

Note that since the conversion rates are numpy arrays, the result is a numpy array with one entry for each arm.
The function *simple_class_profit* is defined as follows:

```python
def simple_class_profit(margin, new_clicks, conversion_rate,
                        future_visits, cost_per_click):
    return new_clicks * (margin * conversion_rate * (1 + future_visits)
                         - cost_per_click)
```

And the estimators $\overline{f_p}$ of the average future visits associated with a price and $\overline{n}$ of the average number of new clicks are computed as follows:

```python
    def compute_future_visits_per_arm(self):
        res = []
        for arm in range(self.n_arms):
            complete_samples = len(self.future_visits_per_arm[arm])
            future_visits = np.sum(self.future_visits_per_arm[arm])
            purchases = np.sum(self.purchases_per_arm[arm][:complete_samples])
            future_visits_per_purchase = future_visits / purchases if purchases else 0
            res.append(future_visits_per_purchase)

        return np.array(res)

    def compute_new_clicks(self):
        return average_ragged_matrix(self.new_clicks_per_arm)
```

It remains to determine how to make an optimistic estimate of the conversion rates: this is left to implement to the subclasses, as in the class *OptimalPriceLearner* the methods are defined as abtract, with the python convention.

```python
    def compute_projection_conversion_rates(self):
        raise NotImplementedError

    def get_average_conversion_rates(self):
        raise NotImplementedError
```

We created two subclasses, one that employs a UCB approach and one that employs a Thompson sampling approach.

4.1 The class *UCBOptimalPriceLearner*

```
def compute_projection_conversion_rates(self):
    return self.compute_conversion_rates_upper_bounds()

def compute_conversion_rates_upper_bounds(self):
    averages = self.compute_conversion_rates_averages()
    radii = self.compute_conversion_rates_radii()
    upper_bounds = averages + radii

    return upper_bounds

def compute_conversion_rates_averages(self):
    return np.array([
        sum(self.purchases_per_arm[arm]) / sum(self.new_clicks_per_arm[arm])
        for arm in range(self.n_arms)
    ]).flatten()

def compute_conversion_rates_radii(self):
    tot_clicks_per_arm = np.array([np.sum(self.new_clicks_per_arm[arm])
                                   for arm in range(self.n_arms)])

    return np.sqrt(2 * np.log(self.current_round) / tot_clicks_per_arm)
```

It can be seen that the average of the conversion rate is estimated, for each arm, as $\mu_a = \frac{purchases_a}{clicks_a}$ and that a confidence bound is computed with the same formula of the UCB1 algorithm for the Bernoulli stochastic bandit environments: $r_a = \sqrt{\frac{2log(t)}{clicks_a}}$. Then, the upper bound $u_a = \mu_a + r_a$ is used as optimistic conversion rate of each arm a to estimate the expected profit in the parent class learning loop.

It should be noticed that since each click is treated as a separate try of a Bernoulli random variable, the updates to the average and to the confidence bounds happen in batches, that is after one round we do not see the realization of one additional try but rather that of new_clicks additional tries of the same arms.

4.2 The class *TSOptimalPriceLearner*

The Thompson sampling learner samples keeps a Beta distribution for each arm and samples the optimistic (or rather, explorative) conversion rate that will be used for that arm:

```
def __init__(self, env: PriceBanditEnvironment):
    super().__init__(env)
    self.beta_cr_priors = [Beta(1, 1, self.env.env.rng) for i in range(self.n_arms)]
```

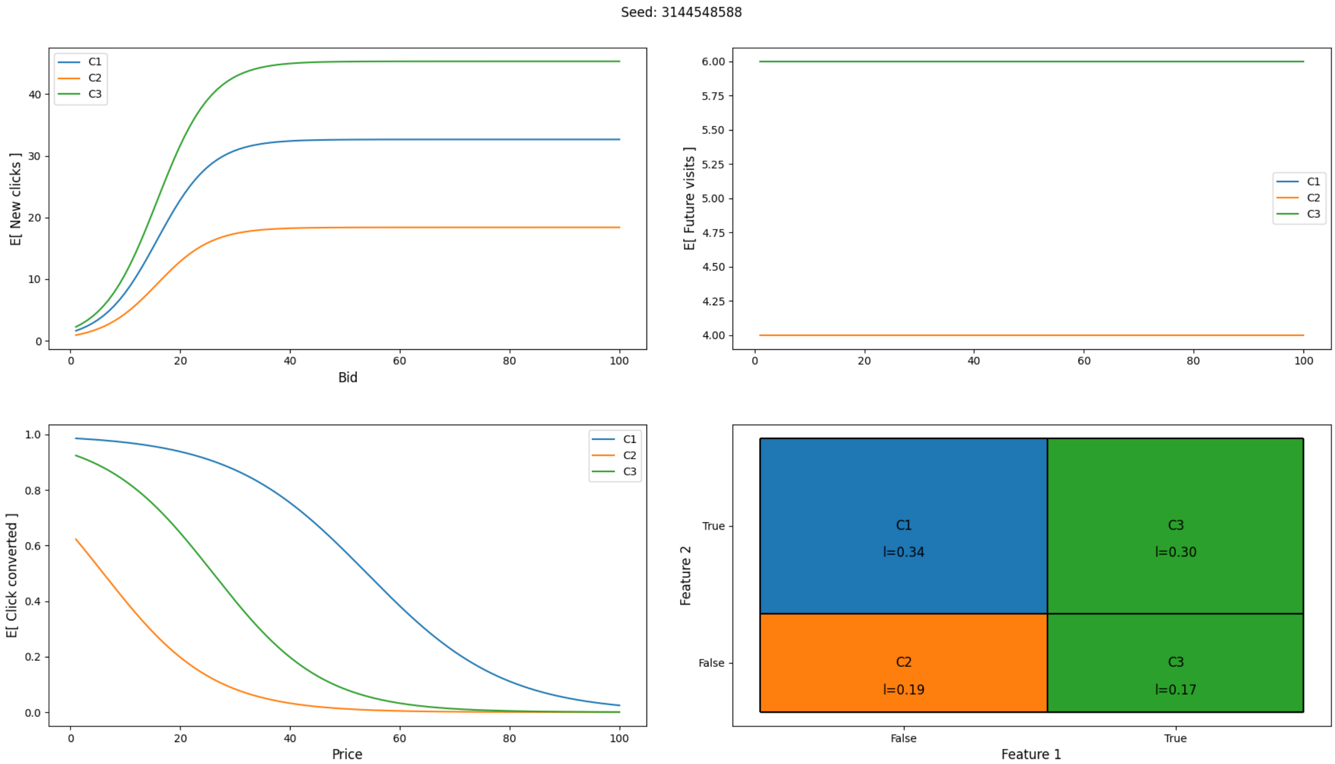
```python
    def compute_projection_conversion_rates(self):
        return self.sample_from_betas()

    def sample_from_betas(self):
        sampled_crs = np.array([b.sample() for b in self.beta_cr_priors]).flatten()
        return sampled_crs
```

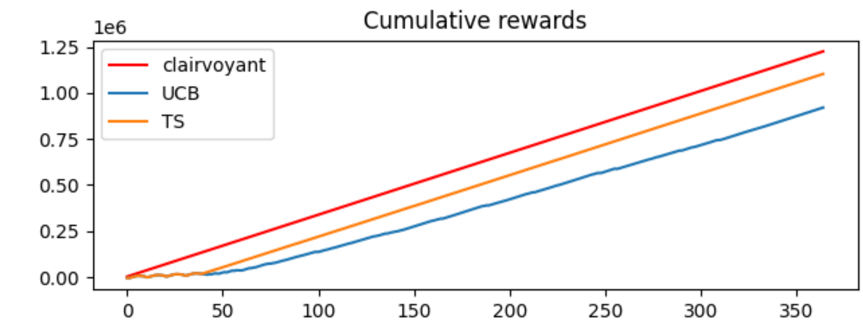
The parent's *pull_from_env(arm)* method is overridden to update the betas whenever new samples are obtained.

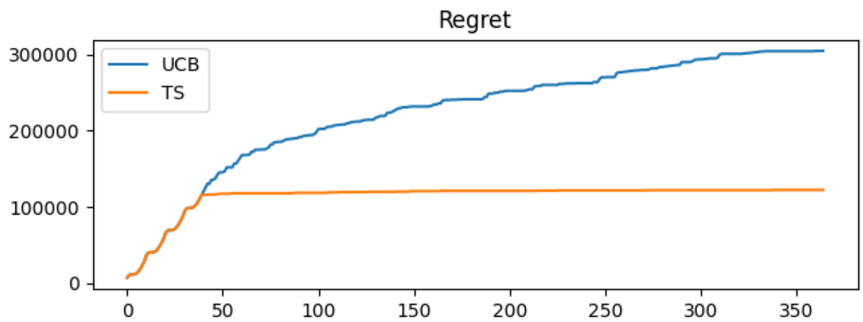
```python
    def pull_from_env(self, arm: int):
        new_clicks, purchases, _, _ = super().pull_from_env(arm)
        self.update_betas(arm, purchases, new_clicks - purchases)

    def update_betas(self, arm: int, successes: int, failures: int):
        self.beta_cr_priors[arm].update_params(successes, failures)
```

It should be noticed that since each click is treated as a separate try of a Bernoulli random variable, the updates to the parameters α and β happen in batches, that is after one round we do not see the realization of one additional try but rather that of *new_clicks* additional tries of the same arm.

4.3 Experiment 1



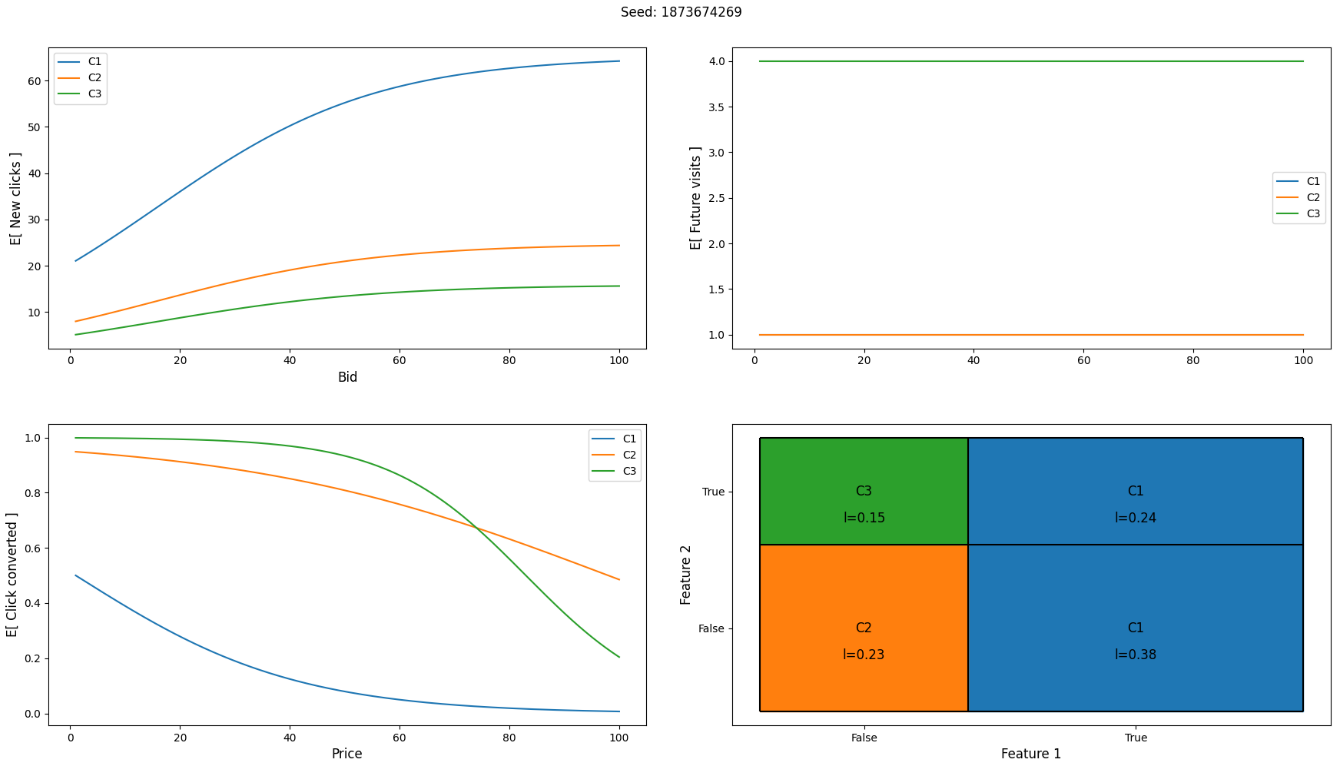


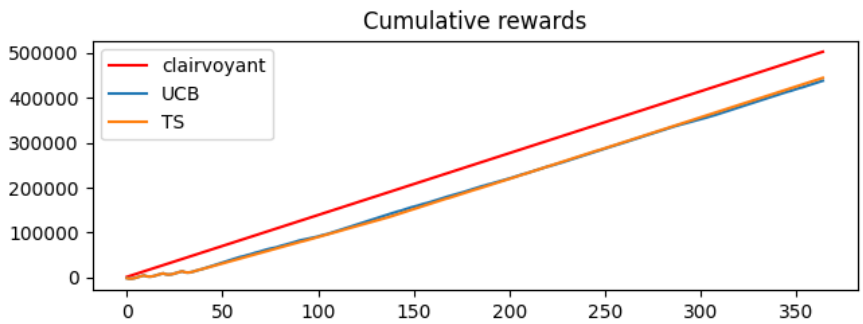


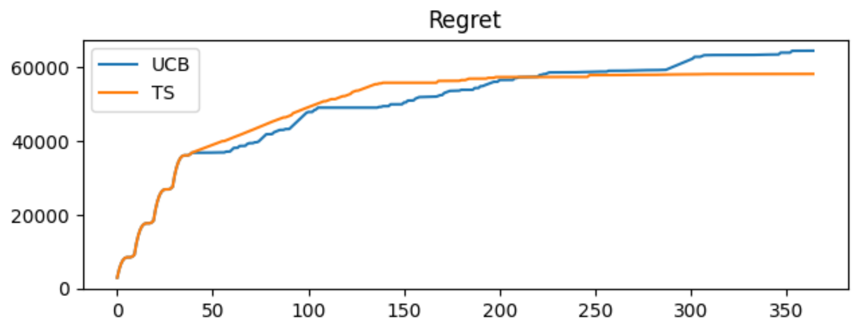
Seed: 3144548588

Price	Expected	Gaps	UCB pulls	UCB expected	TS pulls	TS expected
10.00	-4225.41	7582.43	4	-4187.84	4	-4359.81
20.00	24.99	3332.03	4	137.55	4	109.20
30.00	2527.66	829.35	8	2641.15	4	2139.56
40.00	3357.01	0.00	161	3426.36	309	3420.86
50.00	3043.13	313.88	106	3093.34	24	2955.51
60.00	1993.51	1363.50	38	2155.71	4	1951.62
70.00	704.31	2652.71	10	44.90	4	-147.45
80.00	-365.47	3722.49	17	213.68	4	8.62
90.00	-1068.42	4425.44	6	-1487.12	4	-1413.77
100.00	-1473.00	4830.01	11	-1582.79	4	-1717.06

4.4 Experiment 2







Seed: 1873674269

Price	Expected	Gaps	UCB pulls	UCB expected	TS pulls	TS expected
10.00	-1605.88	2984.16	4	-1768.34	4	-1720.25
20.00	-827.54	2205.82	4	-834.38	4	-813.50
30.00	-177.41	1555.69	4	-191.53	4	-156.00
40.00	370.70	1007.58	4	363.52	4	420.48
50.00	829.98	548.30	5	911.53	5	892.88
60.00	1183.23	195.05	19	1148.35	74	1191.96
70.00	1378.28	0.00	149	1452.68	200	1374.97
80.00	1362.45	15.83	104	1341.69	35	1224.17
90.00	1161.06	217.22	36	1027.17	28	979.59
100.00	888.62	489.65	36	933.26	7	762.71

5 Step 4

In order to perform context generation, the information that the learner receives at each round is divided by combination of features: the method *pull_arm_discriminating(arm_strategy)* accepts a strategy that defines one arm (which means one price) for each combination of features, and returns:

- $n_{TT}, n_{TF}, n_{FT}, n_{FF}$ The new clicks of users of each combination of features.
- $s_{TT}, s_{TF}, s_{FT}, s_{FF}$ The purchases of users of each combination of features.
- $c_{TT}, c_{TF}, c_{FT}, c_{FF}$ The total cost per click paid due to clicks of users of each combination of features.
- $f_{TT}, f_{TF}, f_{FT}, f_{FF}$ The total future purchases of users of each combination of features which visited the website for the first time 30 rounds ago.

Receiving the data in this way, the learner can try any possible strategy and observe the disaggregate result without having access to the underlying class structure. The learner, defined by the class *OptimalPriceDiscriminatingLearner*, keeps track of the context generation thanks to the attribute *context structure*, which is a list of objects of the class *Context*. A *Context* object contains a subset of the combinations of features and the learner initially starts with one context containing all the possible combinations. The estimation of the parameters and of the expected value is performed by each *Context* with the same logic as the non-discriminating learner, by working on the aggregate data aggregated only on its subset of features.
The learning loop is as follows:

```
def learn(self, n_rounds: int):
    self.initial_round_robin()

    while self.current_round < n_rounds:
        self.learn_one_round()

def learn_one_round(self):
    strategy = self.choose_next_strategy()
    self.pull_from_env(strategy=strategy)

    self.update_contexts()
```

The strategy selection must choose one arm for each combination of features: in order to do so, it queries the contexts and each context specifies the next arm for its combinatioins. Let us remind that the contexts always form a partition of the full feature space.

```
def choose_next_strategy(self):
    if self.state_is_explorative_rounds:
        strategy = self.choose_next_strategy_explorative()
```

```
        else:
            strategy = self.choose_next_strategy_normal()
        return strategy

    def choose_next_strategy_normal(self):
        strategy = {}
        for context in self.context_structure:
            arm = context.choose_next_arm(self.new_clicks_per_comb_per_arm,
                                          self.purchases_per_comb_per_arm,
                                          self.tot_cost_per_click_per_comb,
                                          self.future_visits_per_comb_per_arm,
                                          self.current_round)
            for comb in context.features:
                strategy[comb] = arm

        return strategy
```

Let us skip for the moment the code of the method *choose_next_strategy_explorative()*, which is a marginal modification that we have introduced and which we will present later.
One can notice that each context has the method *choose_next_arm(...)*. In this method, each context computes the expected profit with an optimistic / explorative estimate of the conversion rate and returns the arm with the highest projection of profit, estimating all the parameters with aggregated data relative only to the covered set of combinations of features.

```
    def choose_next_arm(self, new_clicks_per_comb_per_arm,
                        purchases_per_comb_per_arm, cost_per_click_per_comb,
                        future_visits_per_comb_per_arm, current_round):
        return int(np.argmax(self.compute_projected_profit(
            new_clicks_per_comb_per_arm,
            purchases_per_comb_per_arm,
            cost_per_click_per_comb,
            future_visits_per_comb_per_arm,
            current_round)
        ))
```

We don't report the code of the method *compute_projected_profit()* since it is identical to the analogous method of the class *OptimalPriceLearner*. Also in this case, the method *compute_projection_conversion_rate()* is defined as abstract to allow for different implementations of explorative strategies.

The second core method is *update_contexts()*:

```
    def update_contexts(self):
        for context in self.context_structure:
```

```
        possible_splits = self.compute_convenient_splits(context)
        if possible_splits:
            incentive, new_structure, feature = min(possible_splits,
                                                    key=lambda x: x[0])
            self.context_structure = new_structure
            print(f'Split context at round {self.current_round} '
                  f'on feature {feature}')
```

For each context, among all the splits that are convenient, it takes the one that is convenient by the smallest amount. The rationale behind this is that it is more likely that the more convenient split will be performed the next round even if a split was just done.

Let's see how the convenient splits are computed:

```
    def compute_convenient_splits(self, context):
        current_lower = self.compute_context_expected_profit_lower_bound(context)
        new_structures = []

        for feature_n, context_true, context_false in self.compute_possible_splits(
                context
        ):
            true_lower = self.compute_context_expected_profit_lower_bound(
                context_true)
            false_lower = self.compute_context_expected_profit_lower_bound(
                context_false)

            incentive = true_lower + false_lower - current_lower
            if incentive > 0:
                new_structure = list(self.context_structure)
                new_structure.remove(context)
                new_structure.append(context_true)
                new_structure.append(context_false)

                print(
                    f'Found convenient split at round {self.current_round} '
                    f'on feature {feature_n}, incentive = {incentive:.2f}')
                new_structures.append((incentive, new_structure, feature_n))
        return new_structures
```

The condition of convenience is that the lower bound of the expected profit after the split is higher that the current lower bound.

For completeness we report also the implementation of the computation of the possible splits:

```
    def compute_possible_splits(self, context):
        n_features = len(self.context_structure[0].features[0])
        res = []
        for i in range(n_features):
            features_where_i_is_true = [f for f in context.features if f[i]]
            features_where_i_is_false = [f for f in context.features if not f[i]]

            # a valid split generates two non-empty contexts
            if features_where_i_is_false and features_where_i_is_true:
                context_true = self.context_creator(features=features_where_i_is_true,
                                                    arm_margin_function=self.env.margin,
                                                    n_arms=self.n_arms,
                                                    rng=self.env.rng)

                context_false = self.context_creator(features=features_where_i_is_false,
                                                     arm_margin_function=self.env.margin
                                                     n_arms=self.n_arms,
                                                     rng=self.env.rng)
                res.append((i, context_true, context_false))

        return res
```

5.1 The class *UCBOptimalPriceDiscriminatingLearner*

Since the logic of the choice of explorative estimates of the conversion rates is performed by the contexts, this class is just a wrapper to instruct the parent class to instantiate *UCBContext* objects when contexts are created.

```
class UCBOptimalPriceDiscriminatingLearner(OptimalPriceDiscriminatingLearner):
    def __init__(self, env: PriceBanditEnvironment):
        super().__init__(env,
                         context_creator=lambda *args,
                                                **kwargs:
                        UCBContext(*args, **kwargs))
```

Here we can see how the *UCBContext* overrides *compute_projection_conversion_rate()*:

```
    def compute_projection_conversion_rate(self, new_clicks_per_arm,
                                           purchases_per_arm, current_round):
        return self._compute_cr_upper_bounds(new_clicks_per_arm,
                                             purchases_per_arm, current_round)

    def _compute_cr_upper_bounds(self, new_clicks_per_arm,
                                 purchases_per_arm, current_round):
        averages = self._compute_cr_averages(new_clicks_per_arm, purchases_per_arm)
```

```
        radia = self.compute_conversion_rates_radia(new_clicks_per_arm, current_round)
        upper_bounds = averages + radia

        return upper_bounds

    def _compute_cr_averages(self, new_clicks_per_arm, purchases_per_arm):
        return np.array([sum(purchases_per_arm[arm]) / sum(new_clicks_per_arm[arm])
                         for arm in range(self.n_arms)]).flatten()

    def compute_conversion_rates_radia(self, new_clicks_per_arm, current_round):
        tot_clicks_per_arm = np.array([np.sum(new_clicks_per_arm[arm])
                                       for arm in range(self.n_arms)])

        return np.sqrt(2 * np.log(current_round) / tot_clicks_per_arm)
```

The confidence bound formula is the same as in *UCBOptimalPriceLearner*.

5.2 The class *TSOptimalPriceDiscriminatingLearner*

We can see the same pattern of the previous class:

```
class TSOptimalPriceDiscriminatingLearner(OptimalPriceDiscriminatingLearner):
    def __init__(self, env: PriceBanditEnvironment):
        super().__init__(env,
                         context_creator=lambda *args,
                                                **kwargs:
                         TSContext(*args, **kwargs))
```

Here we can see how the *TSContext* overrides *compute_projection_conversion_rate()*:

```
    def compute_projection_conversion_rate(self, new_clicks_per_arm,
                                           purchases_per_arm, current_round):
        tot_new_clicks_per_arm = [sum(r) for r in new_clicks_per_arm]
        successes_per_arm = [sum(r) for r in purchases_per_arm]
        failures_per_arm = [tot_new_clicks_per_arm[a] - successes_per_arm[a]
                            for a in range(self.n_arms)]

        betas = [Beta(1 + successes_per_arm[a], 1 + failures_per_arm[a], self.rng)
                 for a in range(self.n_arms)]

        crs = np.array([b.sample() for b in betas]).flatten()

        return crs
```

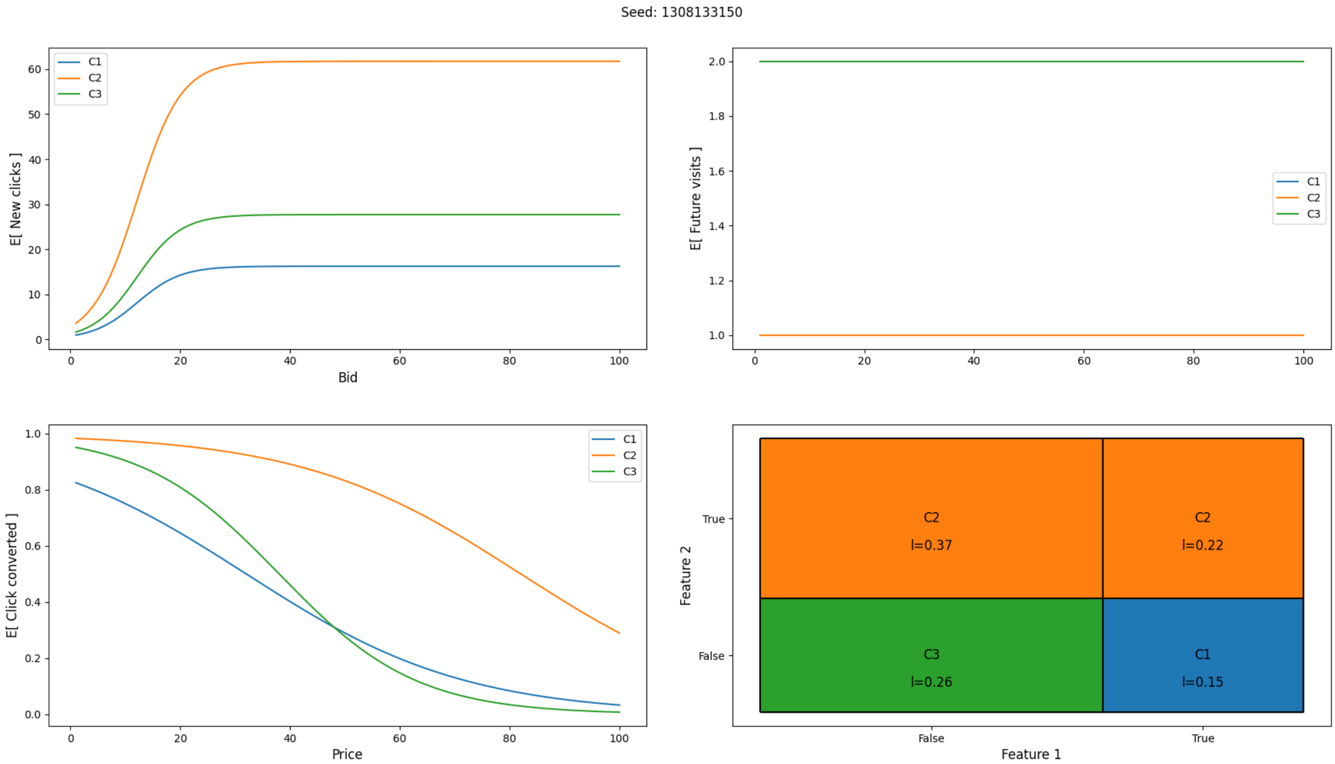
As in Step 3, the Thompson sampling approach samples from the Beta distributions and uses them as the projection values.

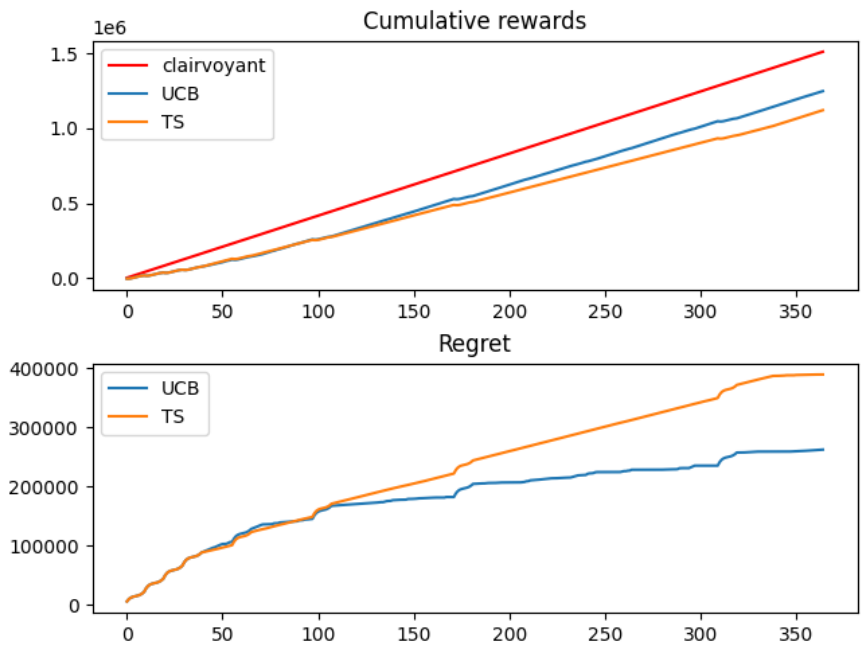
5.3 *choose_next_strategy_explorative()*

We added the explorative rounds after the following empiric observation: the splits that could not be detected after the initial round robin are never detected if their optimal arm is very sub-optimal for the current context, since the lower confidence bound will never get tighter (it will almost never be pulled). Therefore we added the *explorative rounds*, rounds in which the contexts which could be split do not pull the best arm according to their strategy but rather pulll the arms in a round-robin fashion, to allow the confidence bounds of the currently sub-optimal arms to get tighter. We ensure that the number of explorative rounds remains logarithmic in the following way: after the i_{th} explorative round robin has been completed, 2^i normal rounds must be completed before running the $i + 1_{th}$ explorative round robin.

```python
def choose_next_strategy_explorative(self):
    strategy = {}
    for context in self.context_structure:
        could_be_split = len(context.features) > 1

        if could_be_split:
            arm = self.next_round_robin_arm
        else:
            arm = context.choose_next_arm(self.new_clicks_per_comb_per_arm,
                                          self.purchases_per_comb_per_arm,
                                          self.tot_cost_per_click_per_comb,
                                          self.future_visits_per_comb_per_arm,
                                          self.current_round)
        for comb in context.features:
            strategy[comb] = arm

    return strategy
```

5.4 Experiment 1





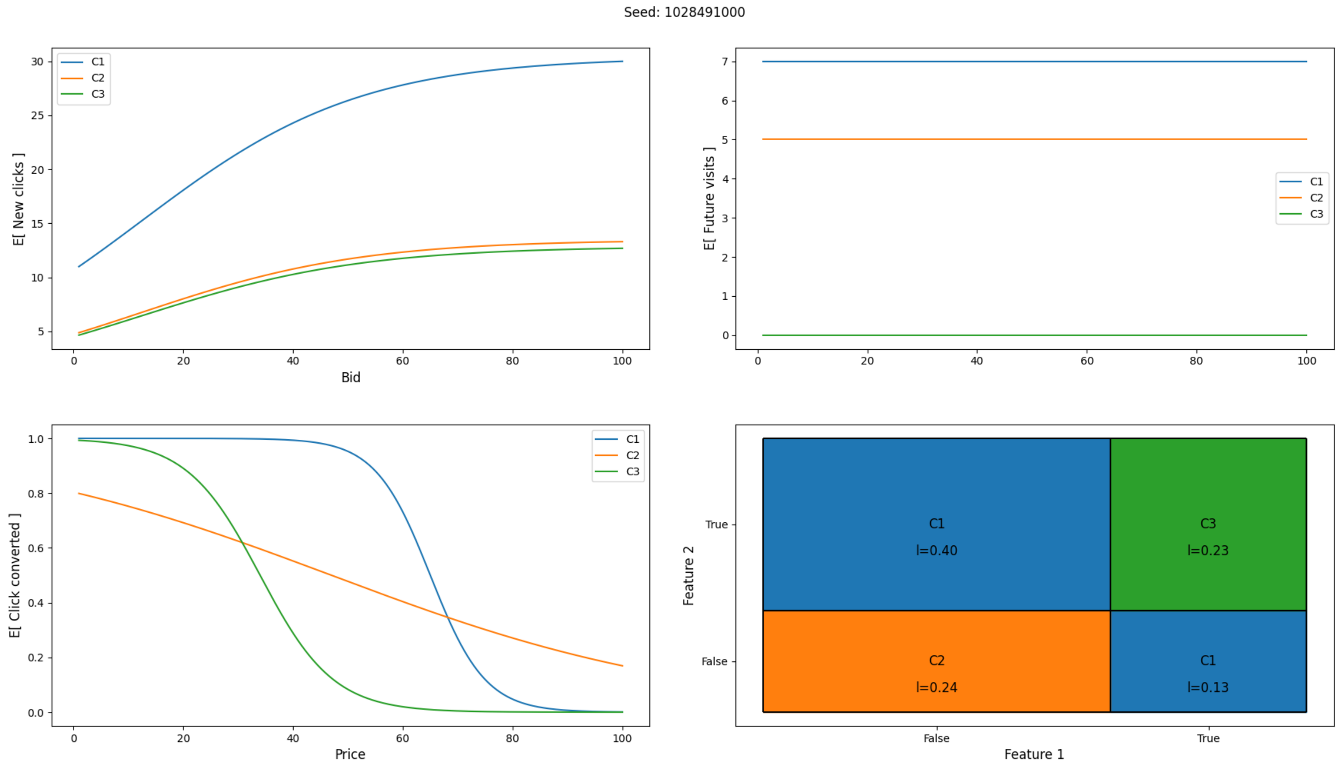
Seed: 1308133150
Legend: Context, true expected value, number of pulls
UCB with context generation:

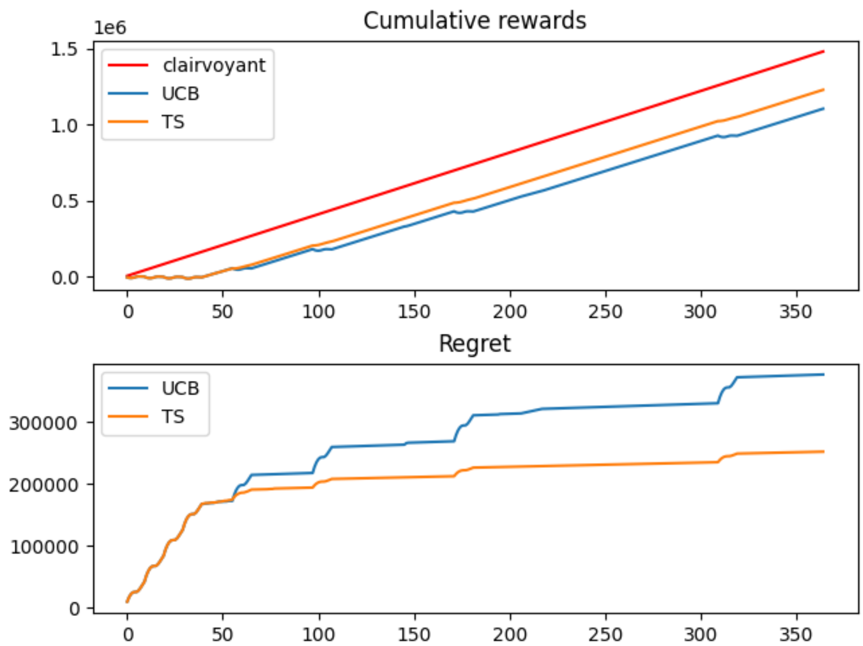
Price	TT,FT	Expected	Pulls	TF,FF	Expected	Pulls
10.00		-1020.63	8		-717.06	8
20.00		44.73	8		172.61	8
30.00		1052.13	8		725.82	24
40.00		1955.04	8		846.51	196
50.00		2684.69	8		621.12	48
60.00		3155.90	56		267.69	8
70.00		3292.43	164		-51.30	22
80.00		3070.78	83		-285.08	17
90.00		2553.79	14		-441.56	15
100.00		1875.71	8		-542.35	19

TS with context generation:

Price	TT,FT	Expected	Pulls	TF	Expected	Pulls	FF	Expected	Pulls
10.00		-1020.63	8		-250.42	8		-466.64	8
20.00		44.73	8		33.32	8		139.29	8
30.00		1052.13	8		210.99	8		514.83	12
40.00		1955.04	8		278.66	12		567.85	30
50.00		2684.69	166		257.68	180		363.43	166
60.00		3155.90	74		184.24	79		83.45	74
70.00		3292.43	68		92.64	43		-143.95	43
80.00		3070.78	9		5.44	8		-290.52	8
90.00		2553.79	8		-67.07	8		-374.49	8
100.00		1875.71	8		-122.69	11		-419.66	8

5.5 Experiment 2





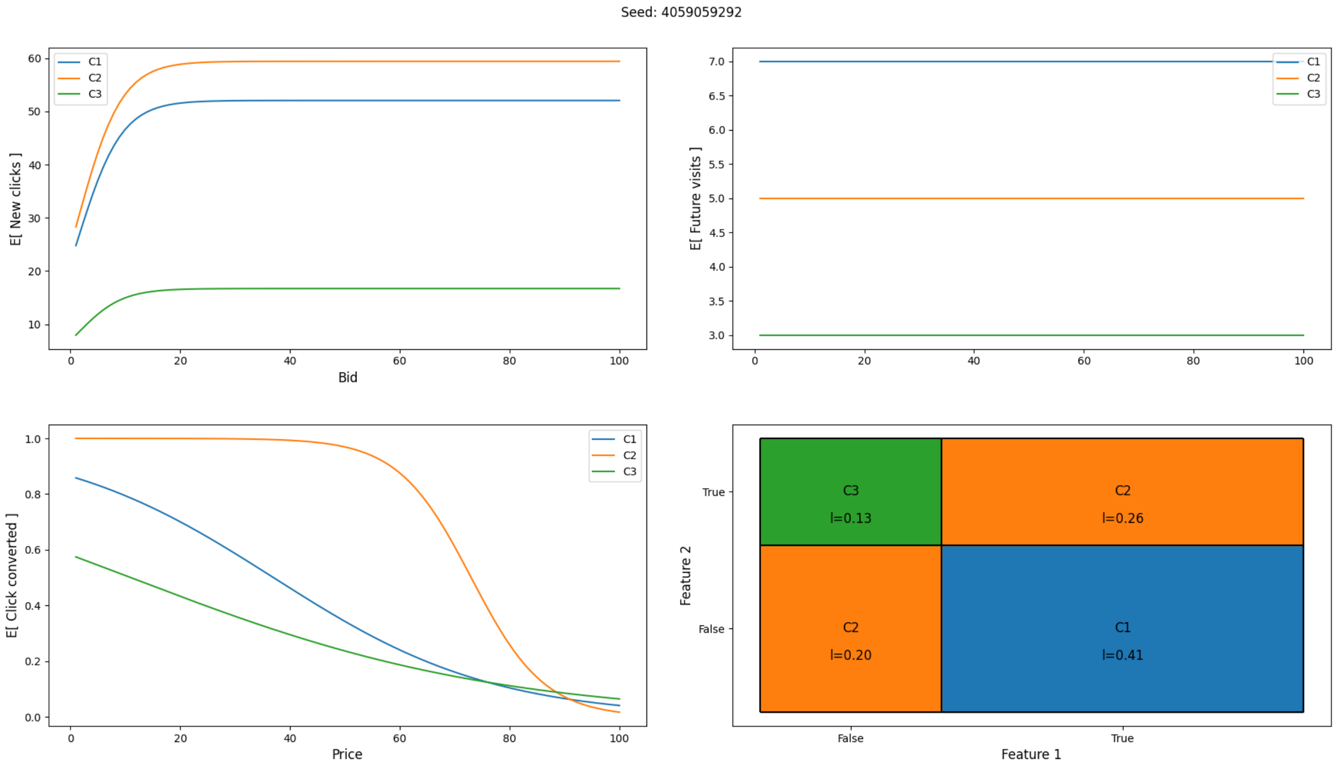
Seed: 1028491000
Legend: Context, true expected value, number of pulls
UCB with context generation:

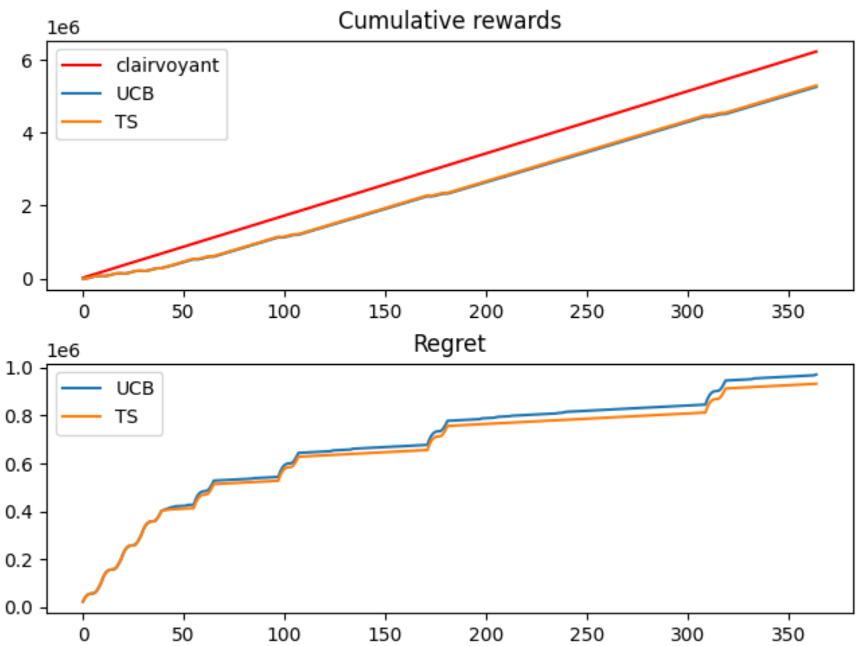
Price	TT,TF,FT,FF	Expected	Pulls
10.00		-5892.08	8
20.00		-3330.67	8
30.00		-871.01	8
40.00		1416.40	8
50.00		3390.47	27
60.00		3959.09	273
70.00		1582.02	9
80.00		-177.67	8
90.00		-633.46	8
100.00		-782.26	8

TS with context generation:

Price	TF,FF	Expected	Pulls	TT	Expected	Pulls	FT	Expected	Pulls
10.00		-2191.47	8		-540.19	4		-3160.42	4
20.00		-1192.25	8		-434.29	4		-1704.14	4
30.00		-272.20	8		-350.35	4		-248.47	4
40.00		552.59	8		-331.65	293		1195.46	4
50.00		1235.59	17		-350.59	14		2505.48	20
60.00		1494.47	282		-363.52	14		2828.13	313
70.00		951.19	10		-368.26	7		999.09	4
80.00		510.92	8		-369.69	7		-318.90	4
90.00		360.16	8		-370.10	9		-623.53	4
100.00		263.18	8		-370.20	9		-675.24	4

5.6 Experiment 2





Seed: 4059059292

Legend: Context, true expected value, number of pulls

UCB with context generation:

Price	TT,TF,FT,FF	Expected	Pulls
10.00		-5800.76	8
20.00		1055.41	8
30.00		6935.22	8
40.00		11663.44	8
50.00		15123.37	15
60.00		16630.06	274
70.00		13915.87	20
80.00		7357.46	8
90.00		2672.17	8
100.00		671.07	8

TS with context generation:

Price	TT,TF,FT,FF	Expected	Pulls
10.00		-5800.76	8
20.00		1055.41	8
30.00		6935.22	8
40.00		11663.44	8
50.00		15123.37	11
60.00		16630.06	290
70.00		13915.87	8
80.00		7357.46	8
90.00		2672.17	8
100.00		671.07	8

6 Step 5

da scrivere

7 Step 6

da scrivere

8 Step 7

da scrivere

A Expected Value of A_{comb}

For each $comb \in \{TT, TF, FT, FF\}$ it holds $\mathbb{E}[A_{comb}] = \lambda_a \tilde{l}_{comb}$

Consider the daily number of auctions A.

$$\mathbb{E}[A_{comb}] = \mathbb{E}[\mathbb{E}[A_{comb}|A]]$$

Since A_{comb} is sampled from a multinomial with A tries and with associated probability $\tilde{l}_{comb}$, it holds

$$\begin{aligned}
\mathbb{E}[A_{comb}] &= \mathbb{E}\left[A\tilde{l}_{comb}\right] \\
&= \mathbb{E}[A]\,\tilde{l}_{comb} \\
&= \lambda_a \tilde{l}_{comb}
\end{aligned}$$

B Expected Value of $N_{c,b}$

Since $\tilde{N}_{comb,b} \sim Binomial(A_{comb}, v(b))$, it holds

$$\begin{aligned}
\mathbb{E}\left[\tilde{N}_{comb,b}\right] &= \mathbb{E}\left[\mathbb{E}\left[\tilde{N}_{comb,b}|A_{comb}\right]\right] \\
&= \mathbb{E}[A_{comb} v(b)] \\
&= \mathbb{E}[A_{comb}]\, v(b) \\
&= \lambda_a \tilde{l}_{comb} v(b)
\end{aligned}$$

therefore

$$\begin{aligned}
\mathbb{E}[N_{c,b}] &= \sum_{comb \in combs(c)} \mathbb{E}\left[\tilde{N}_{comb,b}\right] \\
&= \sum_{comb \in combs(c)} \lambda_a \tilde{l}_{comb} v(b) \\
&= \lambda_a v(b) \sum_{comb \in combs(c)} \tilde{l}_{comb}
\end{aligned}$$

And by definition of the likelihood l_c of a class:

$$\mathbb{E}[N_{c,b}] = \lambda_a v(b) l_c$$

C Derivation of ExpectedProfit

$$
\begin{aligned}
ExpectedProfit(p,b) &= \mathbb{E}\left[\sum_{c\in C}\sum_{i=1}^{N_{c,b}}\Big(D_{c,p,i}(1+F_{c,i})m(p)-C_{c,b,i}\Big)\right]\\
&= \mathbb{E}_{n_{c,b}}\left[\mathbb{E}\left[\sum_{c\in C}\sum_{i=1}^{N_{c,b}}\Big(D_{c,p,i}(1+F_{c,i})m(p)-C_{c,b,i}\Big)\Big|N_{c,b}=n_{c,b}\right]\right]\\
&= \mathbb{E}_{n_{c,b}}\left[\sum_{c\in C}\sum_{i=1}^{n_{c,b}}\Big(\mathbb{E}\left[D_{c,p,i}(1+F_{c,i})m(p)-C_{c,b,i}\Big|N_{c,b}=n_{c,b}\right]\Big)\right]\\
&= \mathbb{E}_{n_{c,b}}\left[\sum_{c\in C}\sum_{i=1}^{n_{c,b}}\Big(\mathbb{E}\left[D_{c,p,i}(1+F_{c,i})m(p)\Big|N_{c,b}=n_{c,b}\right]-k(c,b)\Big)\right]\\
&= \mathbb{E}_{n_{c,b}}\left[\sum_{c\in C}\sum_{i=1}^{n_{c,b}}\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\right]\\
&= \mathbb{E}_{n_{c,b}}\left[\sum_{c\in C}n_{c,b}\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\right]\\
&= \sum_{c\in C}\mathbb{E}_{n_{c,b}}\left[n_{c,b}\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\right]\\
&= \sum_{c\in C}\mathbb{E}_{n_{c,b}}\left[n_{c,b}\right]\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\\
&= \sum_{c\in C}\mathbb{E}\left[N_{c,b}\right]\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\\
&= \sum_{c\in C}n(c,b)\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)
\end{aligned}
$$

D Proof of the Bid Independent Price Hierarchy lemma

$$
\begin{aligned}
ExpectedProfit(p,b) &= \sum_{c\in C}n(c,b)\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)\\
&= \lambda_a v(b)\sum_{c\in C}l_c\Big(r(c,p)(1+f(c))m(p)-k(c,b)\Big)
\end{aligned}
$$

$$
\lambda_a v(b)\sum_{c\in C}l_c\Big(r(c,p_1)(1+f(c))m(p_1)-k(c,b)\Big)\geq\lambda_a v(b)\sum_{c\in C}l_c\Big(r(c,p_2)(1+f(c))m(p_2)-k(c,b)\Big)
$$

Since $\lambda_a > 0$ and $v(b) > 0$:

$$\sum_{c\in C} l_c\Big(r(c,p_1)(1+f(c))m(p_1) - k(c,b)\Big) \geq \sum_{c\in C} l_c\Big(r(c,p_2)(1+f(c))m(p_2) - k(c,b)\Big)$$
$$\sum_{c\in C} l_c r(c,p_1)(1+f(c))m(p_1) - \sum_{c\in C} l_c k(c,b) \geq \sum_{c\in C} l_c r(c,p_2)(1+f(c))m(p_2) - \sum_{c\in C} l_c k(c,b)$$

Since the term $\sum_{c\in C} l_c k(c,b)$ appears on both sides:

$$\sum_{c\in C} l_c r(c,p_1)(1+f(c))m(p_1) \geq \sum_{c\in C} l_c r(c,p_2)(1+f(c))m(p_2)$$

Since all the steps can be reversed with an arbitrary bid b', the relation holds for every possible value of the bid.

E Estimating the future visits per purchase $\overline{f_p}$

Consider the estimation of $\overline{f_{p,b}}$ for a specific pair (p,b) computed before choosing the strategy for round i:

$$R = \{j : p_j = p, b_j = b, j \leq i - 30\}$$
$$\overline{f_{p,b,i}} = \frac{\sum_{j\in R} f_j}{\sum_{j\in R} s_j}$$

Recall that f_j and s_j are the result of a the random experiment of the round j:

$$f_j = \sum_{c\in C} \sum_{k=1}^{N_{c,b_j}} D_{c,p_j,k} F_{c,k}$$
$$\mathbb{E}[f_j] = \sum_{c\in C} n(c,b_j) r(c,p_j) f(c) = \lambda_a v(b_j) \sum_{c\in C} l_c r(c,p_j) f(c)$$
$$s_j = \sum_{c\in C} \sum_{k=1}^{N_{c,b_j}} D_{c,p_j,k}$$
$$\mathbb{E}[s_j] = \sum_{c\in C} n(c,b_j) r(c,p_j) = \lambda_a v(b_j) \sum_{c\in C} l_c r(c,p_j)$$

Given that the learner only sees for each combination (p,b) the total purchases $P_{p,b}$ and the total future visits of people who purchased $F_{p,b}$, the estimator simply computes the empiric mean of $F_{p,b}$ considering each purchase a different sample, that is:

$$\overline{f_{p,b,i}} = \overline{F_{p,b}}_{P_{p,b}}$$
$$\mathbb{E}[\overline{f_{p,b,i}}] = \mathbb{E}[F_{p,b}]$$

Considering only one round, the estimator computes the empiric average of F Therefore at round i:

$$\overline{f_{p,b,i}} = \frac{\sum_{j\in R}\sum_{c\in C}\sum_{k=1}^{N_{c,b,j}} D_{c,p,j,k}F_{c,k}}{\sum_{j\in R}\sum_{c\in C}\sum_{k=1}^{N_{c,b,j}} D_{c,p,j,k}}$$

This estimates $\mathbb{E}[F_p|D_p = 1]$