# Advertising and Pricing

Gabriele Daglio, Federico Di Cesare, Jacopo Germano

July 13, 2021

## 1 The setting

Advertising is used to attract users on an e-commerce 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 probabilities $\theta_1$ and $\theta_2$ 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 modelled:

- 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 e-commerce 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 $\theta_1$ and $\theta_2$. At the generation of the environment, the parameters $\theta_1$ and $\theta_2$ 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 $\lambda_a$ 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 seeing it 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 sigmoid function for this purpose:

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

where $\overline{z} > 0$ and $\overline{b} > 0$ 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 belonging to 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) sigmoid 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 modelled 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. We assume that the tax domicile of the e-commerce is located in the Cayman Islands, therefore the owner pays no taxes whatsoever, and 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 $p$.

## 1.7 Simulation of one day

Once the *Environment* has been created, it is wrapped by one of the *BanditEnvironment*. The learner, in order to simulate one day of selling, pulls an arm from a *BanditEnvironment* which, in turn, calls the method *simulate_one_day(pricing_strategy, bidding_strategy)* exposed by its internal *Environment*.

```
def simulate_one_day(self, pricing_strategy, bidding_strategy):
    purchases, tot_cost, new_future_visits, new_clicks = {}, {}, {}, {}

    auctions, new_clicks = self.distNewClicks.sample_bidding_strategy(bidding_strategy)
    profit = 0

    for c in self.classes:
        for comb in c.features:
            price = pricing_strategy[comb]
            bid = bidding_strategy[comb]
            purchases[comb] = self.distClickConverted.sample_n(c, price, new_clicks[comb])
            tot_cost[comb] = sum(self.distCostPerClick.sample_n(c, bid, new_clicks[comb]))
            new_future_visits[comb] = sum(self.distFutureVisits.sample_n(c, purchases[comb]))
            profit += self.margin(price) * (purchases[comb] + new_future_visits[comb]) - tot_cost[comb]

    return auctions, new_clicks, purchases, tot_cost, new_future_visits, profit
```

First, the *Environment* samples the total number of auctions and the new clicks for each combinations of users' features, given the bidding strategy chosen by the learner. Then, again for each combination, given its price and bid strategies, it computes the purchases, the total cost of the clicks and the future visits. Eventually, it computes the overall profit according to the chosen arms.

## 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* $ExpectedProfit(p_1,\overline{b}) \geq ExpectedProfit(p_2,\overline{b})$ *for some bid value* $\overline{b}$*, 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:

1. 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 $O(|P|)$
2. 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 $O(|B|)$
3. Returns the solution $(p^*, b^*)$.

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

The implementation is as follows:
We first define the function *ExpectedProfit(p,b)*:

```
def expected_profit(env, p, b, classes=None):
    m = env.margin
    n = env.distNewClicks.mean
    r = env.distClickConverted.mean
    f = env.distFutureVisits.mean
    k = env.distCostPerClick.mean

    C = classes if classes is not None else env.classes

    profit = sum(
        simple_class_profit(m(p), n(c, b), r(c, p), f(c), k(c, b))
        for c in C
    )

    return profit
```

Where the function *simple_class_profit(...)* is defined as:

```
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)
```

The algorithm is as follows:

```
def step1(env, prices, bids):
    median_b = bids[len(bids) // 2]

    optimal_price = optimal_price_for_bid(env, prices, median_b)
    optimal_bid = optimal_bid_for_price(env, bids, optimal_price)

    profit = expected_profit(env, optimal_price, optimal_bid)

    return optimal_price, optimal_bid, profit
```

The methods *optimal_price_for_bid(b)* and *optimal_bid_for_price(p)* are implemented as follows:

```
def optimal_price_for_bid(env, prices, bid, classes=None):
    opt_p_index = np.argmax([
        expected_profit(env, p, bid, classes)
        for p in prices
    ])

    return prices[opt_p_index]


def optimal_bid_for_price(env, bids, price, classes=None):
    opt_b_index = np.argmax([
        expected_profit(env, price, b, classes)
        for b in bids
    ])
    return bids[opt_b_index]
```

## 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 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:

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

- 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 on both $p$ 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. 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 those 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, which corresponds to our implementation.*

# 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):
    average_new_clicks = self.compute_average_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()
    tot_clicks = sum_ragged_matrix(self.new_clicks_per_arm)
    cost_per_click = self.tot_cost / tot_clicks
```

```
    projected_profit = simple_class_profit(
        margin=margin, conversion_rate=crs, new_clicks=average_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 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:

```
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_average_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 abstract, with the python convention.

```
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*

We implemented the UCB approach by sub-classing *OptimalPriceLearner* and overriding the following methods:

```
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)
```

The average of the conversion rate is estimated, for each arm, as $\mu_a = \frac{purchases_a}{clicks_a}$ and the confidence bound radius 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 arm.

### 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:

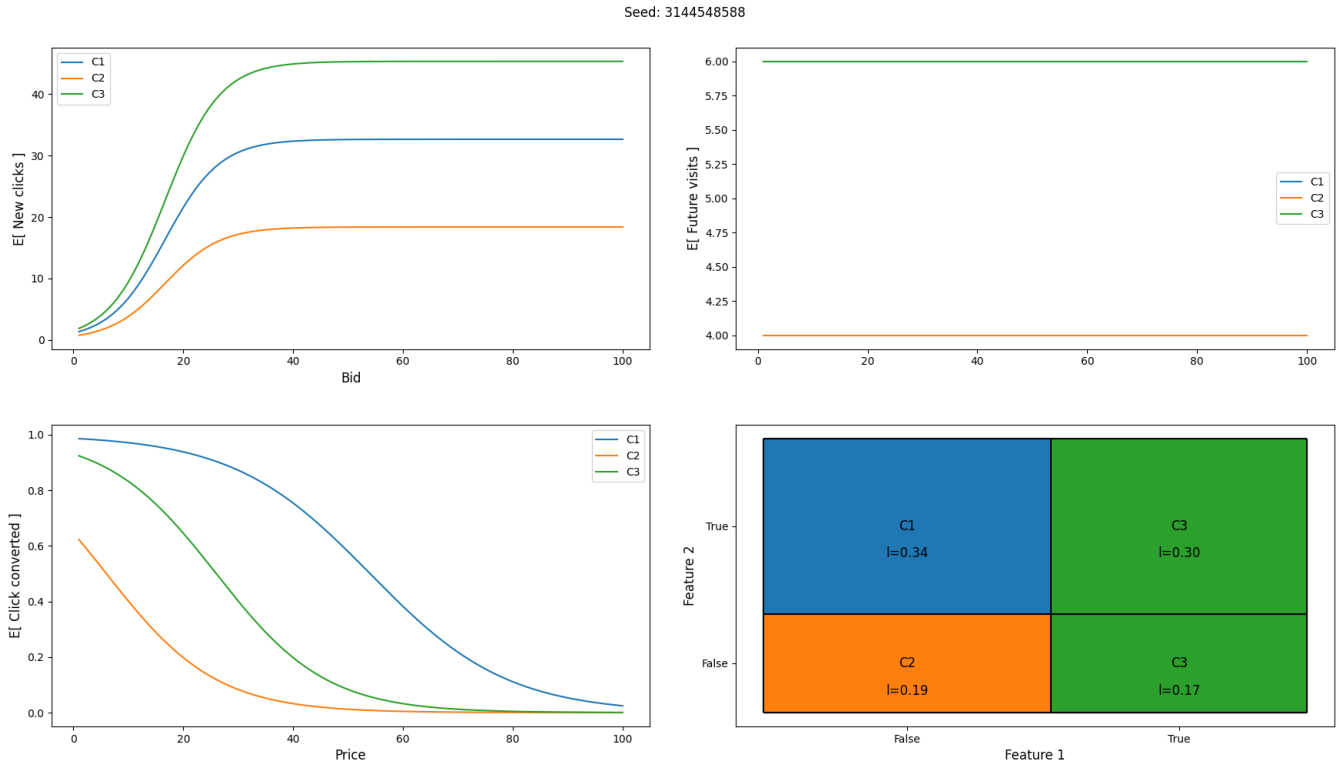
```
def __init__(self, env: PriceBanditEnvironment):
    super().__init__(env)
    self.beta_cr_priors = [Beta(1, 1, self.env.env.rng) for i in range(self.n_arms)]

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
```

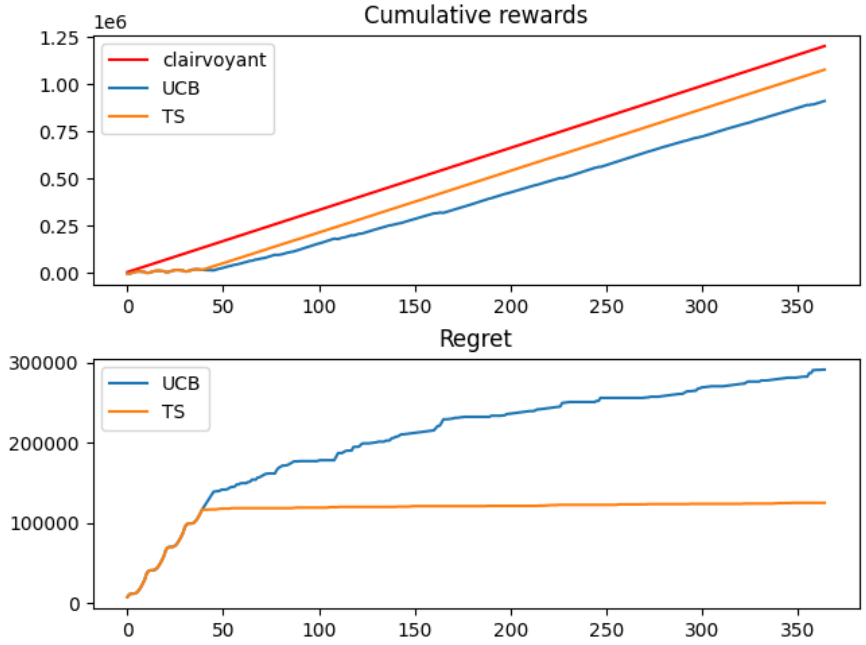
The parent's *pull_from_env(arm)* method is overridden to update the betas whenever new samples are obtained.

```
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 $\alpha$ and $\beta$ 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



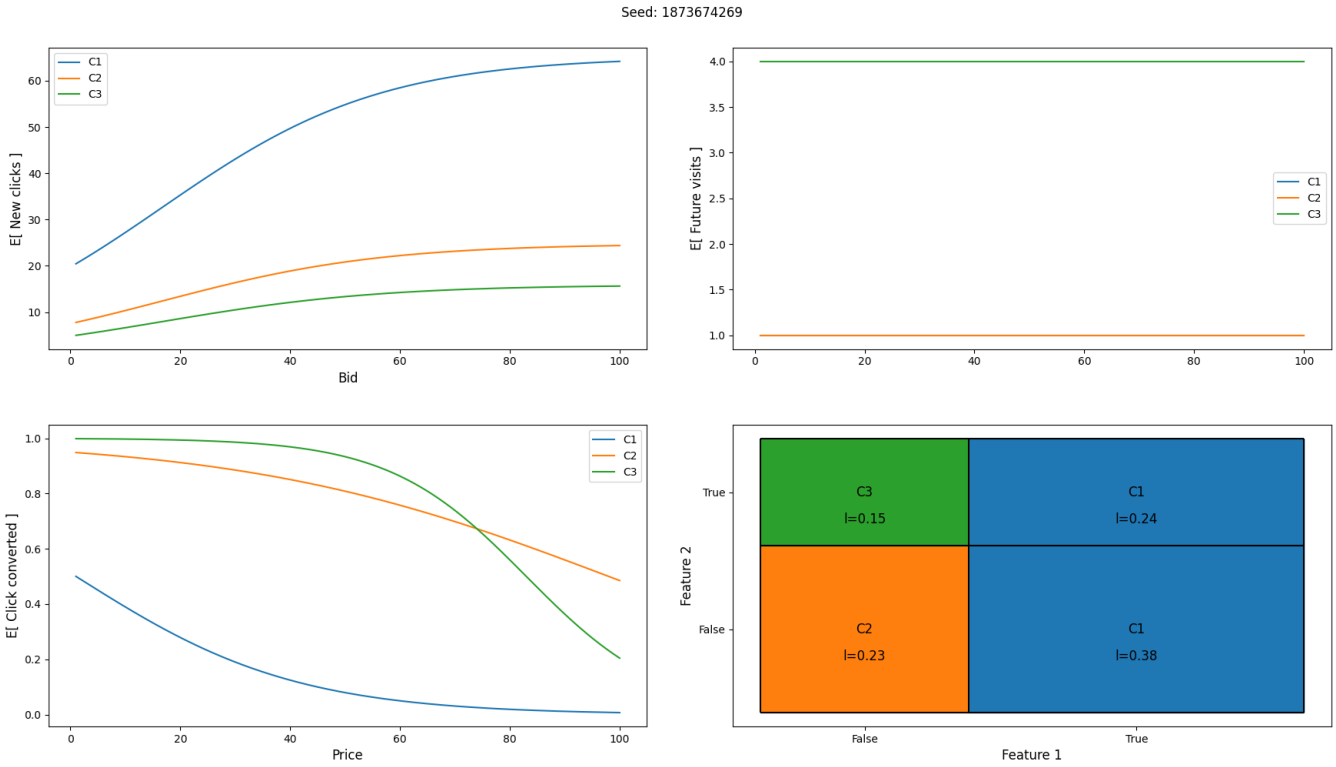



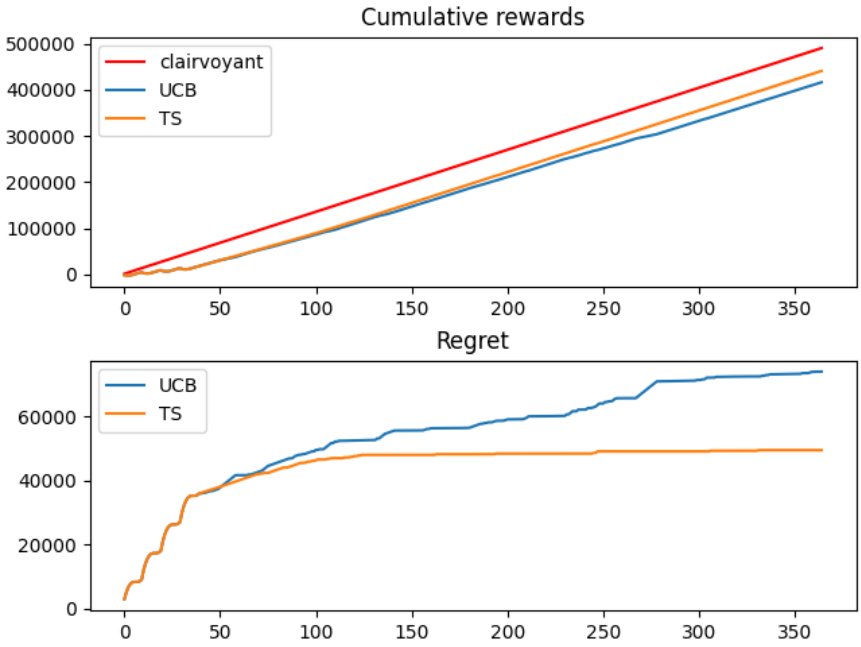
Seed: 3144548588

| Price | Expected | Gaps | UCB Pulls | UCB Expected | TS Pulls | TS Expected |
|---|---|---|---|---|---|---|
| 10.00 | -4309.67 | 7601.39 | 4 | -4287.03 | 4 | -4447.41 |
| 20.00 | -48.64 | 3340.36 | 4 | 16.98 | 4 | 28.71 |
| 30.00 | 2460.29 | 831.43 | 10 | 2650.07 | 4 | 2066.38 |
| 40.00 | 3291.72 | 0.00 | 141 | 3266.95 | 302 | 3344.78 |
| 50.00 | 2977.05 | 314.67 | 131 | 3045.23 | 31 | 3016.02 |
| 60.00 | 1924.81 | 1366.91 | 34 | 2020.80 | 4 | 1859.43 |
| 70.00 | 632.38 | 2659.34 | 13 | 218.62 | 4 | -180.93 |
| 80.00 | -440.08 | 3731.80 | 13 | -435.83 | 4 | -165.58 |
| 90.00 | -1144.79 | 4436.51 | 11 | -1236.55 | 4 | -1492.98 |
| 100.00 | -1550.37 | 4842.09 | 4 | -1806.97 | 4 | -1715.24 |

Optimal price: 40.00, Optimal bid: 27.00

## 4.4 Experiment 2



Seed: 1873674269

| Price | Expected | Gaps | UCB Pulls | UCB Expected | TS Pulls | TS Expected |
|---|---|---|---|---|---|---|
| 10.00 | -1564.32 | 2906.91 | 4 | -1677.11 | 4 | -1704.78 |
| 20.00 | -806.12 | 2148.72 | 4 | -775.53 | 4 | -778.56 |
| 30.00 | -172.82 | 1515.42 | 4 | -200.15 | 4 | -153.46 |
| 40.00 | 361.10 | 981.50 | 4 | 264.64 | 4 | 414.62 |
| 50.00 | 808.49 | 534.11 | 5 | 765.44 | 4 | 993.70 |
| 60.00 | 1152.60 | 190.00 | 14 | 1044.73 | 27 | 1177.04 |
| 70.00 | 1342.60 | 0.00 | 95 | 1351.48 | 261 | 1351.32 |
| 80.00 | 1327.18 | 15.42 | 122 | 1336.60 | 7 | 825.18 |
| 90.00 | 1131.00 | 211.59 | 65 | 1138.87 | 45 | 1097.34 |
| 100.00 | 865.62 | 476.98 | 48 | 967.43 | 5 | 666.85 |

Optimal price: 70.00, Optimal bid: 9.00

# 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 combinations. 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_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, tot_cost_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,
        tot_cost_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:
        convenient_splits = self.compute_convenient_splits(context)
        if convenient_splits:
            incentive, new_structure, feature = max(convenient_splits,
                                                    key=lambda x: x[0])
            self.context_structure = new_structure
            self.performed_splits.append((self.current_round, feature, incentive))
            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 largest amount.

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 = []

    possible_splits = self.compute_possible_splits(context)

    for feature_n, context_true, context_false in possible_splits:
        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):
        combinations_where_i_is_true = [f for f in context.features if f[i]]
        combinations_where_i_is_false = [f for f in context.features if not f[i]]

        # a valid split generates two non-empty context
        if combinations_where_i_is_false and combinations_where_i_is_true:
            context_true = self.context_creator(features=combinations_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=combinations_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)
    radii = self.compute_conversion_rates_radii(new_clicks_per_arm, current_round)
    upper_bounds = averages + radii

    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_radii(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, **
```

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
```

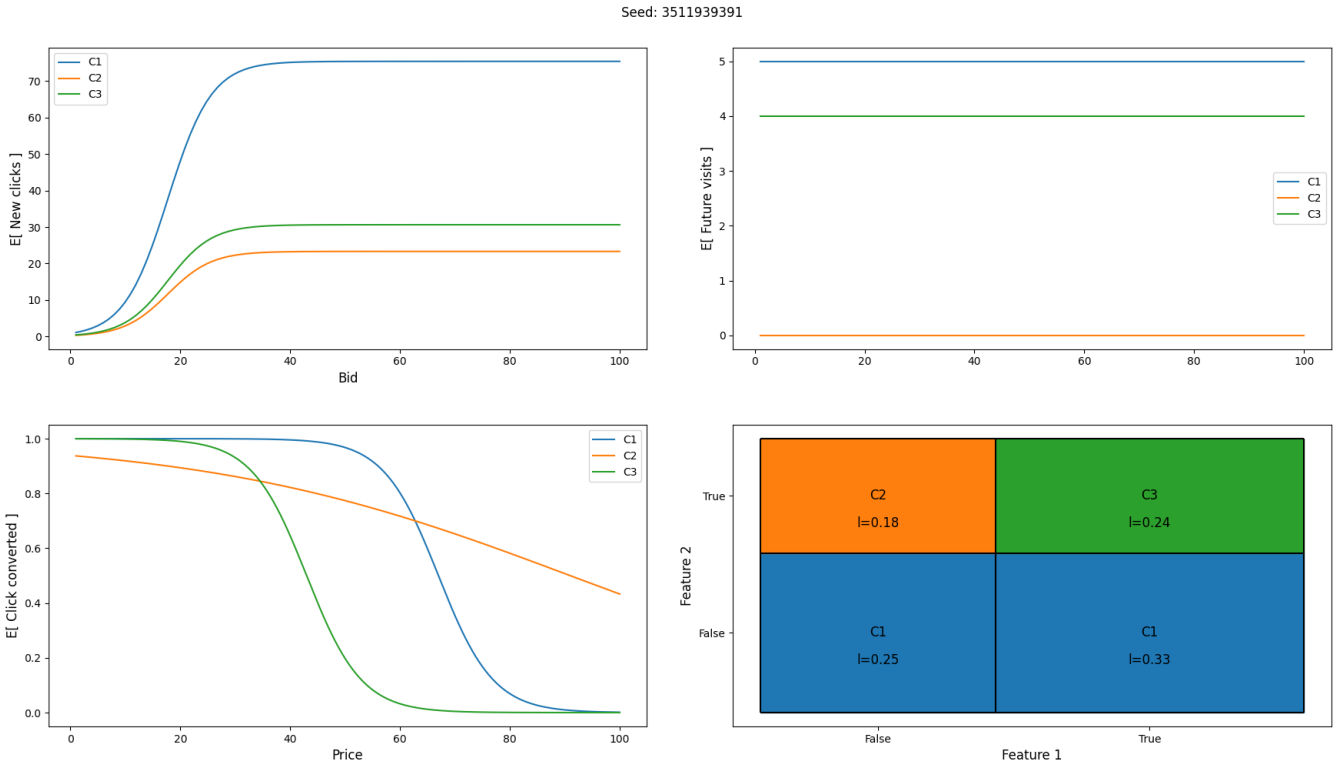
As in Step 3, the Thompson sampling approach samples from the Beta distributions and uses them as the projection values.

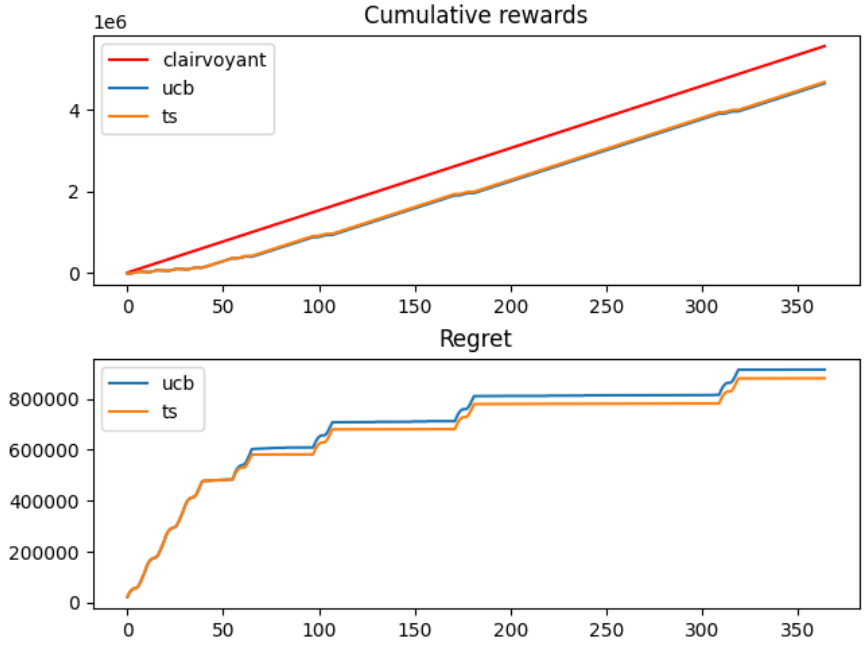
### 5.3 *choose_next_strategy_explorative()*

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 pull 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. From the regret plots below, one can clearly see the explorative rounds as the regret makes a step.

```
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_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: 3511939391
 Legend: context, true expected value, number of pulls
 UCB with context generation:
```

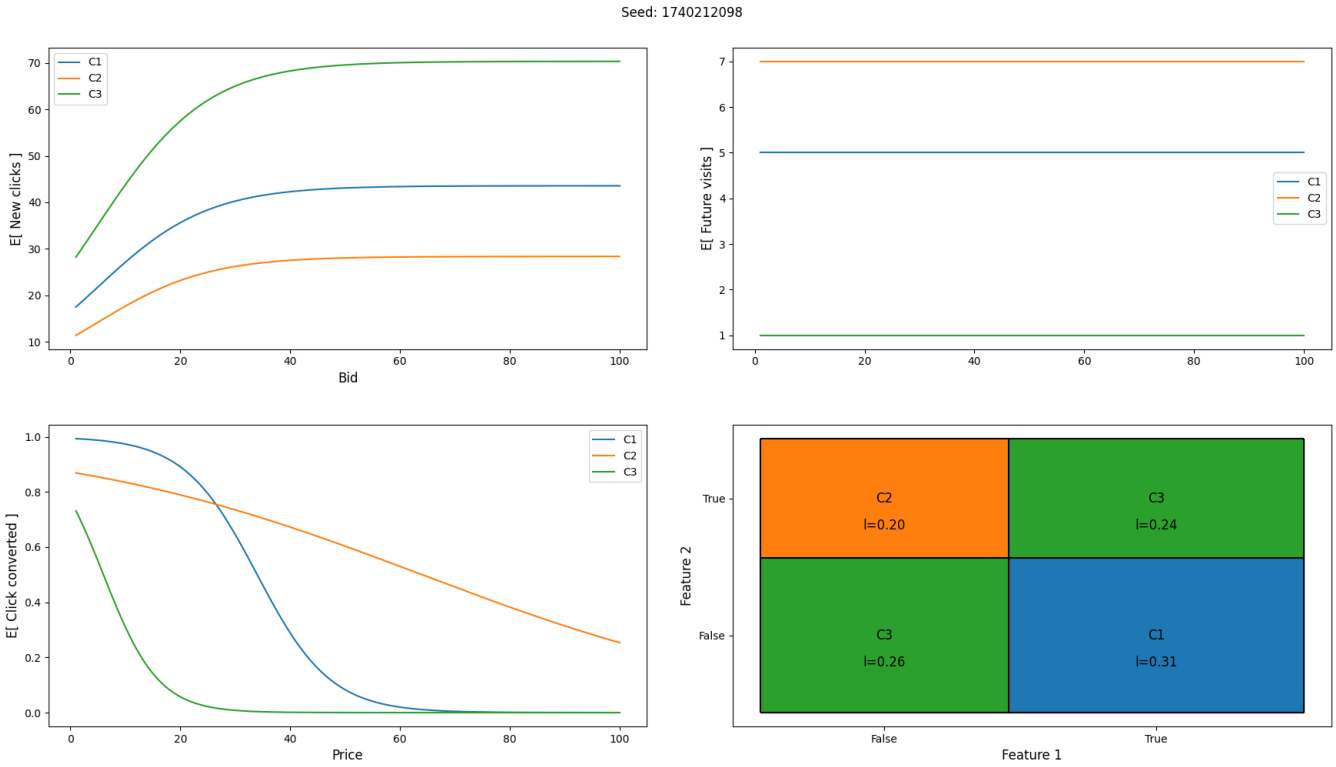
| Price | TF,FF | Expected | Pulls | TT | Expected | Pulls | FT | Expected | Pulls |
|---|---|---|---|---|---|---|---|---|---|
| 10.00 | | -4517.03 | 8 | | -1720.20 | 5 | | -683.40 | 5 |
| 20.00 | | -153.91 | 8 | | -249.24 | 5 | | -478.75 | 5 |
| 30.00 | | 4205.80 | 8 | | 1091.63 | 8 | | -287.62 | 5 |
| 40.00 | | 8525.74 | 8 | | 1455.78 | 315 | | -115.08 | 32 |
| 50.00 | | 12457.01 | 14 | | 159.53 | 6 | | 33.10 | 5 |
| 60.00 | | 13501.64 | 287 | | -624.54 | 6 | | 151.11 | 8 |
| 70.00 | | 6449.89 | 8 | | -798.68 | 5 | | 234.04 | 42 |
| 80.00 | | 31.61 | 8 | | -828.79 | 5 | | 279.03 | 43 |
| 90.00 | | -1577.76 | 8 | | -833.67 | 5 | | 286.28 | 143 |
| 100.00 | | -1849.29 | 8 | | -834.43 | 5 | | 259.47 | 77 |

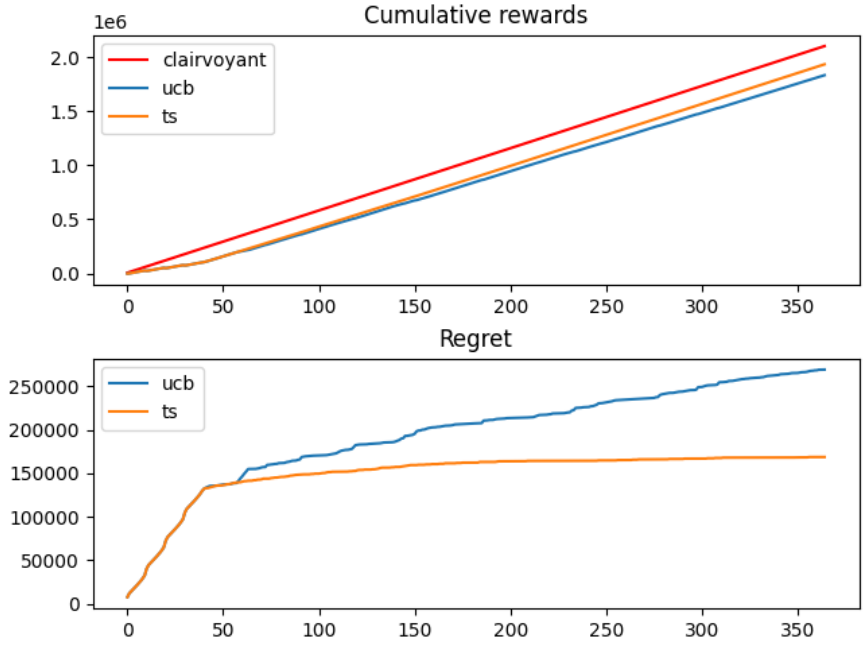
```
 Performed splits:
 Round 41 split on feature 2 with incentive 1147.21
 Round 78 split on feature 1 with incentive 32.89

 TS with context generation:
```

| Price | TF,FF | Expected | Pulls | TT | Expected | Pulls | FT | Expected | Pulls |
|---|---|---|---|---|---|---|---|---|---|
| 10.00 | | -4517.03 | 8 | | -1720.20 | 4 | | -683.40 | 4 |
| 20.00 | | -153.91 | 8 | | -249.24 | 4 | | -478.75 | 4 |
| 30.00 | | 4205.80 | 8 | | 1091.63 | 4 | | -287.62 | 4 |
| 40.00 | | 8525.74 | 8 | | 1455.78 | 328 | | -115.08 | 17 |
| 50.00 | | 12457.01 | 8 | | 159.53 | 4 | | 33.10 | 4 |
| 60.00 | | 13501.64 | 293 | | -624.54 | 5 | | 151.11 | 5 |
| 70.00 | | 6449.89 | 8 | | -798.68 | 4 | | 234.04 | 17 |
| 80.00 | | 31.61 | 8 | | -828.79 | 4 | | 279.03 | 99 |
| 90.00 | | -1577.76 | 8 | | -833.67 | 4 | | 286.28 | 53 |
| 100.00 | | -1849.29 | 8 | | -834.43 | 4 | | 259.47 | 158 |

```
 Performed splits:
 Round 41 split on feature 2 with incentive 2994.85
 Round 54 split on feature 1 with incentive 122.42


Optimal pricing strategy: C1(TF, FF): 60.00, C2(FT): 90.00, C3(TT): 40.00
```

## 5.5 Experiment 2



```
 Seed: 1740212098
 Legend: context, true expected value, number of pulls
 UCB with context generation:
```

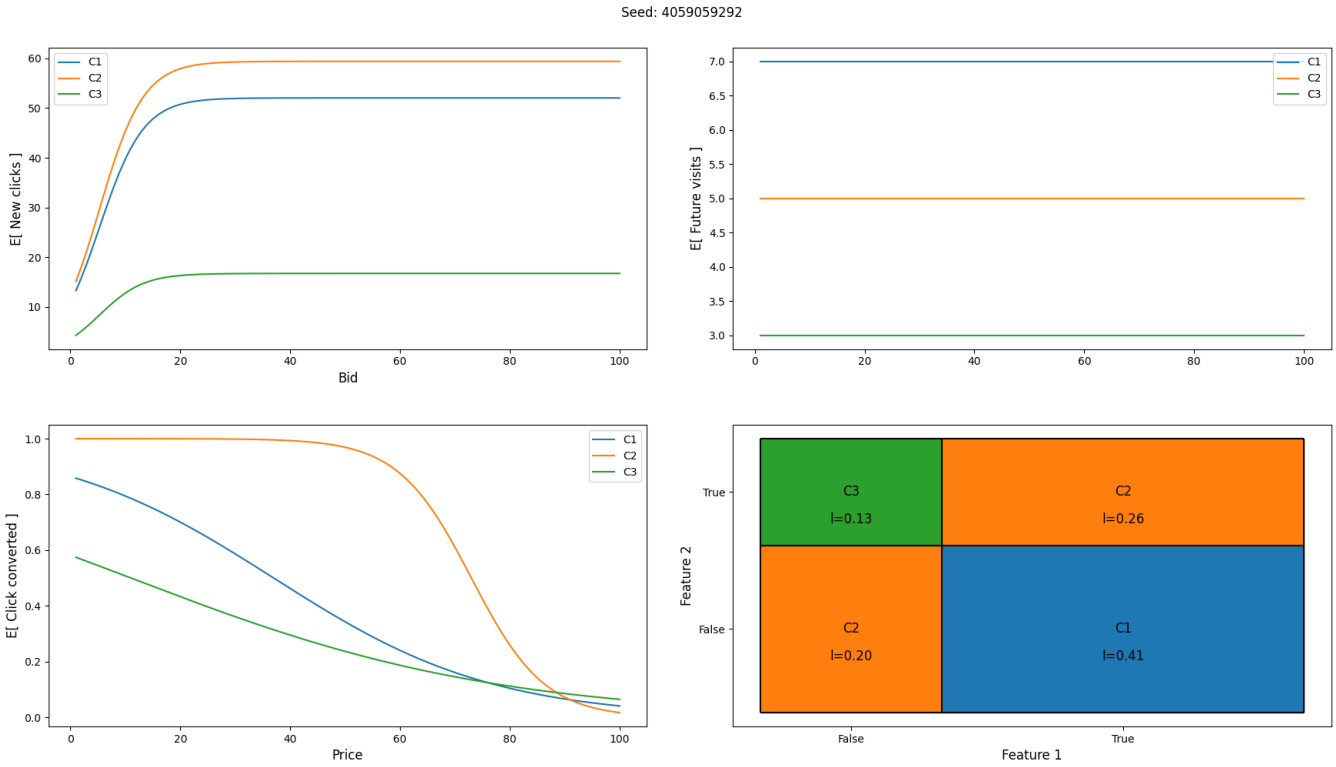
| Price | TF | Expected | Pulls | FF | Expected | Pulls | TT | Expected | Pulls | FT | Expected | Pulls |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | | -721.78 | 4 | | -357.94 | 4 | | -330.17 | 4 | | -405.27 | 4 |
| 20.00 | | 1084.30 | 4 | | -311.49 | 4 | | -287.33 | 6 | | 978.13 | 4 |
| 30.00 | | 1937.84 | 297 | | -331.78 | 4 | | -306.05 | 12 | | 2174.06 | 4 |
| 40.00 | | 1157.48 | 17 | | -338.69 | 7 | | -312.42 | 10 | | 3139.81 | 4 |
| 50.00 | | 125.29 | 6 | | -340.17 | 125 | | -313.79 | 20 | | 3842.23 | 12 |
| 60.00 | | -330.98 | 7 | | -340.45 | 23 | | -314.04 | 30 | | 4266.16 | 30 |
| 70.00 | | -472.97 | 4 | | -340.49 | 31 | | -314.08 | 44 | | 4420.21 | 91 |
| 80.00 | | -512.30 | 18 | | -340.50 | 43 | | -314.09 | 59 | | 4337.12 | 89 |
| 90.00 | | -522.69 | 4 | | -340.50 | 54 | | -314.09 | 79 | | 4068.00 | 100 |
| 100.00 | | -525.36 | 4 | | -340.50 | 70 | | -314.09 | 101 | | 3672.50 | 27 |

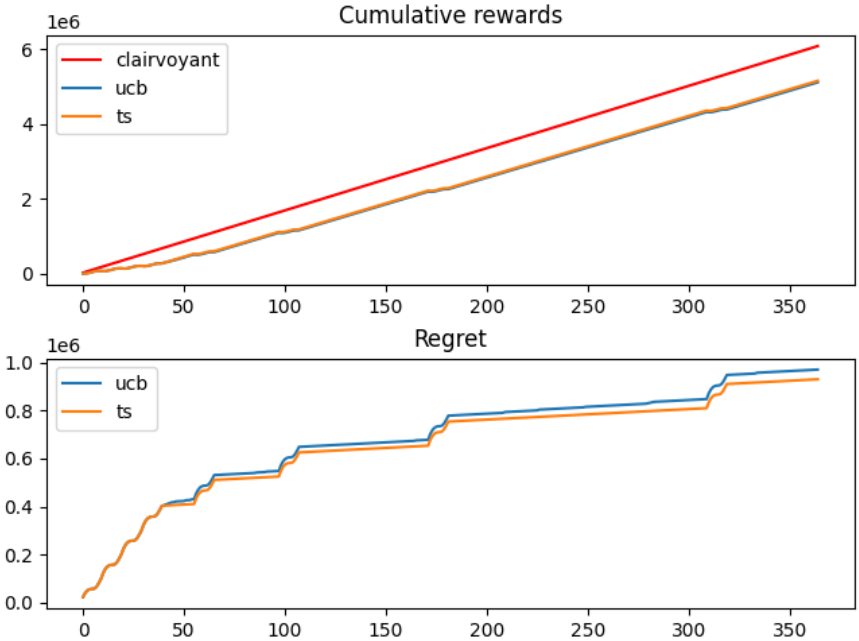
```
 Performed splits:
 Round 41 split on feature 2 with incentive 2983.76
 Round 42 split on feature 1 with incentive 149.32
 Round 44 split on feature 1 with incentive 434.91

 TS with context generation:
```

| Price | TT | Expected | Pulls | FT | Expected | Pulls | TF | Expected | Pulls | FF | Expected | Pulls |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | | -330.17 | 4 | | -405.27 | 4 | | -721.78 | 4 | | -357.94 | 4 |
| 20.00 | | -287.33 | 263 | | 978.13 | 4 | | 1084.30 | 4 | | -311.49 | 262 |
| 30.00 | | -306.05 | 11 | | 2174.06 | 4 | | 1937.84 | 328 | | -331.78 | 27 |
| 40.00 | | -312.42 | 19 | | 3139.81 | 4 | | 1157.48 | 4 | | -338.69 | 5 |
| 50.00 | | -313.79 | 8 | | 3842.23 | 15 | | 125.29 | 4 | | -340.17 | 10 |
| 60.00 | | -314.04 | 11 | | 4266.16 | 11 | | -330.98 | 5 | | -340.45 | 7 |
| 70.00 | | -314.08 | 11 | | 4420.21 | 245 | | -472.97 | 4 | | -340.49 | 10 |
| 80.00 | | -314.09 | 10 | | 4337.12 | 6 | | -512.30 | 4 | | -340.50 | 13 |
| 90.00 | | -314.09 | 13 | | 4068.00 | 60 | | -522.69 | 4 | | -340.50 | 13 |
| 100.00 | | -314.09 | 15 | | 3672.50 | 12 | | -525.36 | 4 | | -340.50 | 14 |

```
 Performed splits:
 Round 41 split on feature 2 with incentive 5179.85
 Round 42 split on feature 1 with incentive 2768.04
 Round 43 split on feature 1 with incentive 78.48


Optimal pricing strategy: C1(TF): 30.00, C2(FT): 70.00, C3(TT, FF): 20.00
```

## 5.6 Experiment 3



```
 Seed: 4059059292
 Legend: context, true expected value, number of pulls
 UCB with context generation:
+--------+-------------+----------+-------+
|  Price | TT,TF,FT,FF | Expected | Pulls |
+--------+-------------+----------+-------+
|  10.00 |             | -6199.86 |     8 |
|  20.00 |             |   659.73 |     8 |
|  30.00 |             |  6542.47 |     8 |
|  40.00 |             | 11273.06 |     8 |
|  50.00 |             | 14734.71 |    16 |
|  60.00 |             | 16242.16 |   274 |
|  70.00 |             | 13526.62 |    19 |
|  80.00 |             |  6964.93 |     8 |
|  90.00 |             |  2277.29 |     8 |
| 100.00 |             |   275.20 |     8 |
+--------+-------------+----------+-------+

 Performed splits:

 TS with context generation:
+--------+-------------+----------+-------+
|  Price | TT,TF,FT,FF | Expected | Pulls |
+--------+-------------+----------+-------+
|  10.00 |             | -6199.86 |     8 |
|  20.00 |             |   659.73 |     8 |
|  30.00 |             |  6542.47 |     8 |
|  40.00 |             | 11273.06 |     8 |
|  50.00 |             | 14734.71 |     9 |
|  60.00 |             | 16242.16 |   292 |
|  70.00 |             | 13526.62 |     8 |
|  80.00 |             |  6964.93 |     8 |
|  90.00 |             |  2277.29 |     8 |
| 100.00 |             |   275.20 |     8 |
+--------+-------------+----------+-------+

 Performed splits:


Optimal pricing strategy: C1(TF): 50.00, C2(TT, FF): 60.00, C3(FT): 60.00
```

# 6 Step 5

In order to learn online the best bid for a fixed price, we implement a symmetric approach with respect to Step 3, with which it has many similarities. We defined the class *BidBanditEnvironment*, which has internally an instance of *Environment*. Upon the creation of an instance of a *BidBanditEnvironment*, the underlying *Environment*, the set of possible bids and the fixed price value must be supplied as arguments. The bandit environment hides the actual bids $B$ from the learner, and instead it shows a bandit-like set of arms numbered from 0 to $|B| - 1$.

The *BidBanditEnvironment* exposes the method *pull_arm_not_discriminating(arm: int)* which returns to the aggregated data:
*auctions, new_clicks, purchases, tot_cost_per_clicks, (past_arm, past_future_visits).*
We can notice that the number of auctions run by the advertiser is also communicated to the learner, differently from the *PriceBanditEnvironment* of Step 3.

The *OptimalBidLearner* is also similar to the learner of Step 3, it is evident from the core learning loop:

```
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):
    mask = self.compute_safe_arms()
    # put compute_projected_profits where mask is true and 0 otherwise
    arms_bid_safe = np.where(mask, self.compute_projected_profits(), 0)
    return int(np.argmax(arms_bid_safe))
```

The arm selection logic is, as always, to pull the arm with the highest expected profit, computed with an optimistic / explorative estimate of the parameter (in this case, the probability of winning an auction with a certain bid value). However, here we notice a difference: a safety constraint is introduced to prevent the learner from playing arms that would yield a negative profit with a probability greater or equal to a certain threshold. We call this threshold the *security* parameter, which is set by default to 0.2.

The set of safe arms is computed as follows:

```
def compute_safe_arms(self):
    means = [np.mean(new_clicks) for new_clicks in self.new_clicks_per_arm]
    std_dev = [np.std(new_clicks) for new_clicks in self.new_clicks_per_arm]

    lower_security_value = [norm.ppf(self.security, m, std)
                            for m, std in zip(means, std_dev)]
    expected_profits = self.compute_expected_profits(nc=lower_security_value)
    arm_mask = expected_profits > 0

    return arm_mask
```

A safe arm is defined as an arm which revenue won't be negative with a certain probability. To compute the set of safe arms we perform normal regression with the data of our random variable (the new clicks) and assuming we not to sell the product below cost we can take the 20th percentile of the distribution and compute the expected profit with that value. This way we are sure that no more than the 20% of the times we will have a lower profit. The security parameters regulates how much "safe" we want to be. The computation of the projection of profit for each arm is as follows:

```
def compute_projected_profits(self):
    auctions = self.compute_average_auctions()
    winning_probs = self.compute_projection_auction_winning_probability_per_arm()
    new_clicks = auctions * winning_probs
    # constant margin of the fixed price
    margin = self.env.margin()
    crs = self.compute_conversion_rates()
    future_visits = self.compute_future_visits()
    cost_per_click = np.array(
        [self.tot_cost_per_arm[arm] / np.sum(self.new_clicks_per_arm[arm])
         for arm in range(self.n_arms)]
    )

    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
```

The estimated quantities are computed in the following way:

```
def compute_average_auctions(self):
    return average_ragged_matrix(self.auctions_per_arm)

def compute_conversion_rates(self):
    return sum_ragged_matrix(self.purchases_per_arm) / \
           sum_ragged_matrix(self.new_clicks_per_arm)

def compute_future_visits(self):
    successes_sum = 0

    for arm in range(self.n_arms):
        complete_samples = len(self.future_visits_per_arm[arm])
        successes_sum += np.sum(self.purchases_per_arm[arm][:complete_samples])

    if not successes_sum:
        return 0
    else:
        return sum_ragged_matrix(self.future_visits_per_arm) / successes_sum
```

While the method *compute_projection_auction_winning_probability_per_arm()* is defined as abstract to allow for the implementation of different explorative strategies.

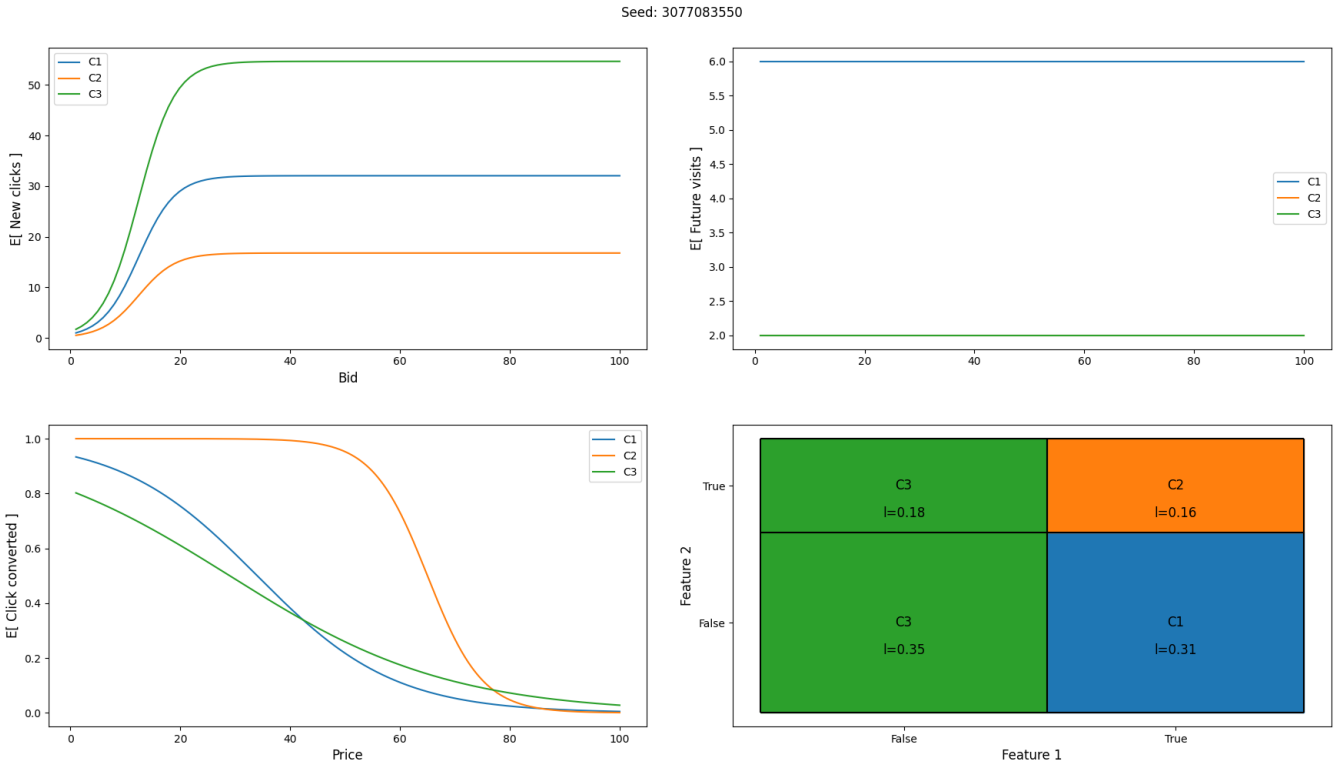
## 6.1 The class *UCBOptimalBidLearner*

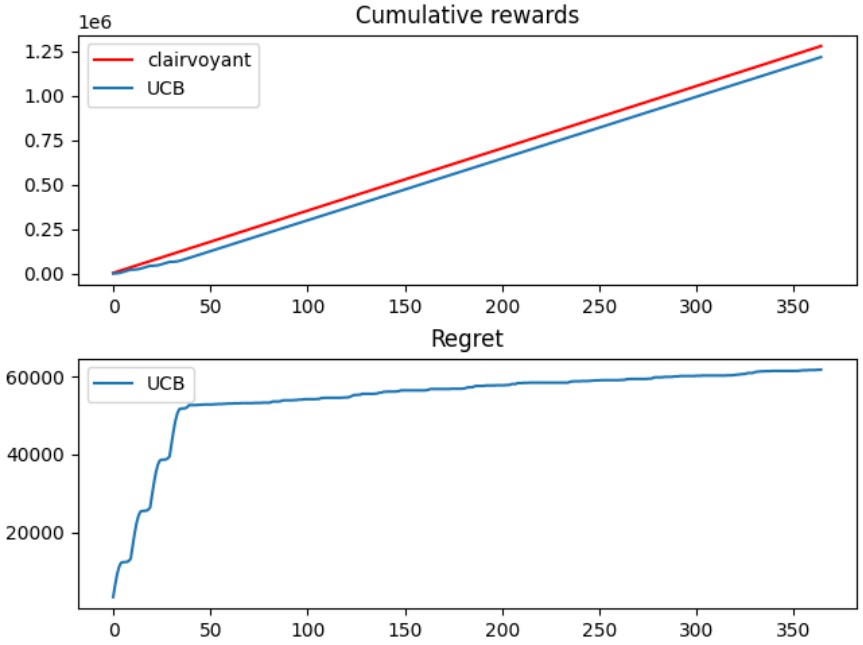
We have chosen the UCB approach to continue the assignment, the upper confidence bounds for the probability of winning an auction with a given bid value are computed in the following way:

```
def compute_projection_auction_winning_probability_per_arm(self):
    avg = self.compute_average_auction_winning_probability_per_arm()
    radii = self.compute_auction_winning_probability_radii()
    upper_bounds = avg + radii

    return upper_bounds

def compute_auction_winning_probability_radii(self):
    auctions_per_arm = np.array([sum(self.auctions_per_arm[a])
                                 for a in range(self.n_arms)])
    return np.sqrt(2 * np.log(self.current_round) / auctions_per_arm)
```

The bound is the standard upper confidence bound used by UCB1 for stochastic Bernoulli bandits, where each auction is considered one try and the click of the user (which under our assumption is equivalent to winning the auction) is the success.

## 6.2 Experiment 1



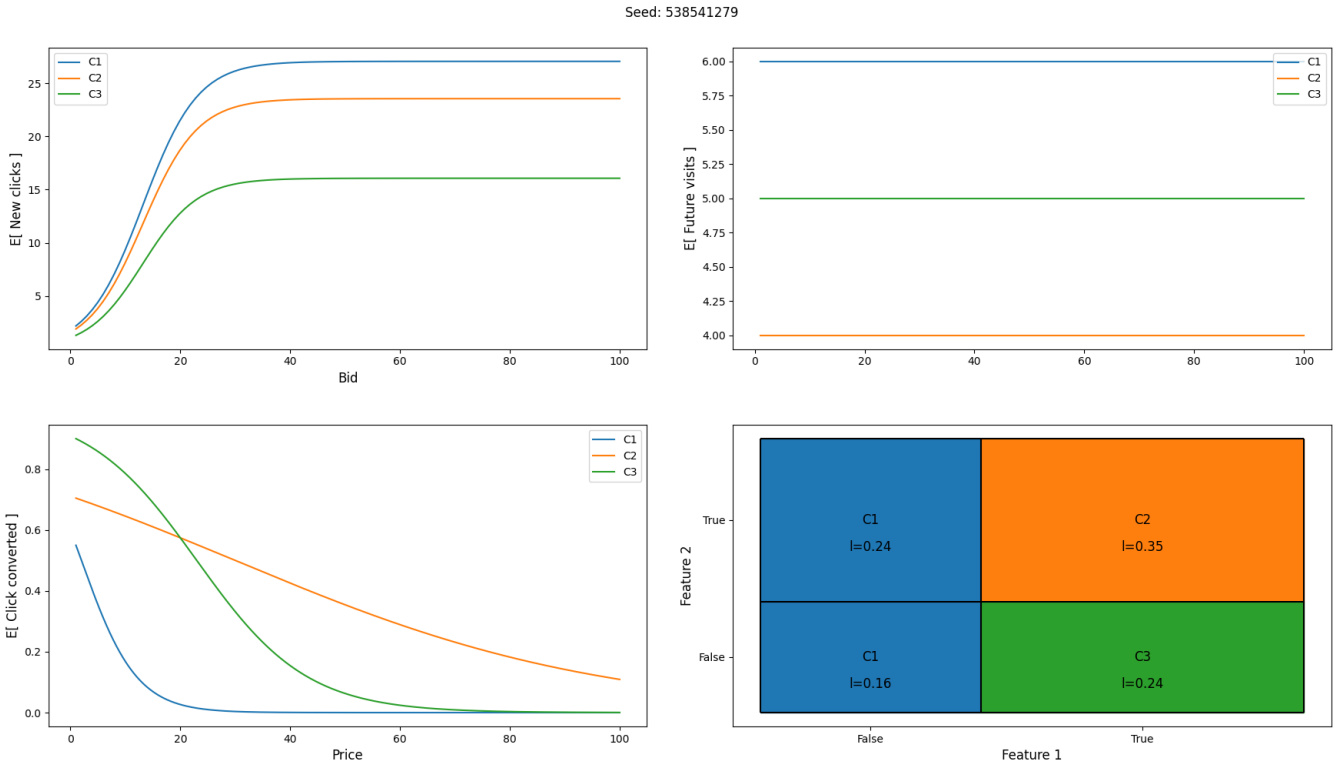



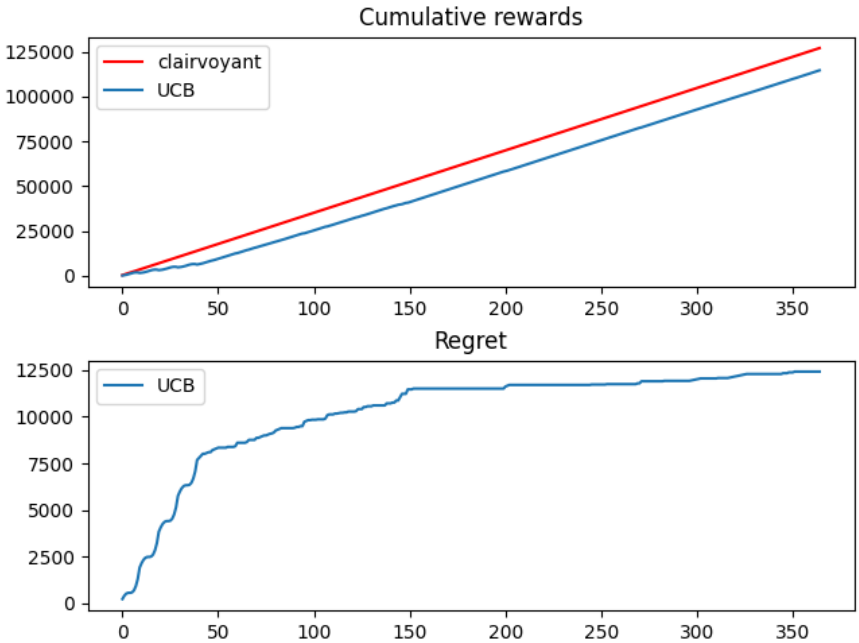
Seed: 3077083550

| Bid | True expected | Gaps | Pulls | Learner expected |
|---|---|---|---|---|
| 1.00 | 176.27 | 3327.71 | 4 | 221.40 |
| 4.00 | 394.24 | 3109.74 | 4 | 365.96 |
| 7.00 | 830.11 | 2673.87 | 4 | 860.37 |
| 10.00 | 1557.46 | 1946.52 | 4 | 1601.24 |
| 13.00 | 2446.80 | 1057.18 | 4 | 2370.49 |
| 17.00 | 3302.31 | 201.67 | 15 | 3224.52 |
| 20.00 | 3503.98 | 0.00 | 226 | 3506.75 |
| 23.00 | 3446.78 | 57.20 | 87 | 3462.11 |
| 26.00 | 3261.94 | 242.04 | 13 | 3279.35 |
| 30.00 | 2937.62 | 566.36 | 4 | 2931.16 |

Optimal price: 42.00, Optimal bid: 20.00

## 6.3 Experiment 2



Seed: 538541279

| Bid | True expected | Gaps | Pulls | Learner expected |
|---|---|---|---|---|
| 1.00 | 112.66 | 235.42 | 6 | 92.84 |
| 4.00 | 168.78 | 179.30 | 8 | 162.38 |
| 7.00 | 237.41 | 110.68 | 17 | 229.92 |
| 10.00 | 305.21 | 42.88 | 39 | 281.62 |
| 13.00 | 348.09 | 0.00 | 247 | 322.53 |
| 17.00 | 326.47 | 21.61 | 32 | 304.58 |
| 20.00 | 243.51 | 104.58 | 4 | 192.83 |
| 23.00 | 116.61 | 231.48 | 4 | 75.33 |
| 26.00 | -36.98 | 385.06 | 4 | -70.67 |
| 30.00 | -261.17 | 609.26 | 4 | -290.20 |

Optimal price: 57.00, Optimal bid: 13.00

# 7 Step 6

In this step we have to learn in online fashion both the price and the bid, while not discriminating among the customer classes. In other words we will work only with aggregate data and we will provide a single optimal tuple of arms. The rationale behind our approach is the following: we still have two random variables, the conversion rate (pricing problem) and the probability of winning an auction (bidding problem). All the other quantities involved in the problem depend only on one of the two random variables. We have to store our data in a matrix, with the two axis being the prices and the bids, but since a quantity will depend only on one single axis, during the learning we can project this quantity on the other axis and thus exploit the functions we wrote before for the disjointed problems.
The environment is an instance of *JointBanditEnvironment* which is pretty much the same as *PriceBanditEnvironment* and *BidBanditEnvironment*, but the method *pull_arm_not_discriminating* accepts both a price arm and a bid arm. As for the delayed feedback, being a joint problem environment, this class will return the future visits with a tuple of arms, not with a single arm as before.
The learning loop is the very same as the previous steps so there is no point in reporting it here again, what is interesting is the routine for choosing and arm. Here we exploit the results from Step 1, where we stated that the optimal price is constant for all the bids. Indeed to choose the next tuple of arms we firstly compute the price arm we want to pull with a fixed bid, then given that price arm, we compute the bid arm we want to pull.

```
def choose_next_arm(self):
    median_bid = self.n_arms_bid // 2
    arm_price = np.argmax(self.compute_projected_profits_fixed_bid(median_bid))
    mask = self.compute_safe_arms(arm_price)
    arms_bid_safe = np.where(mask,
                             self.compute_projected_profits_fixed_price(arm_price), 0)
    arm_bid = np.argmax(arms_bid_safe)
    return arm_price, arm_bid
```

Note that there still is the safety constraint introduced at step 5.
Since the choice of the arms is divided in two steps, we have the two functions to select the most promising arm, namely *compute_projected_profits_fixed_bid* and *compute_projected_profits_fixed_price* which are reported below.

```
def compute_projected_profits_fixed_bid(self, arm_bid):
    new_clicks = self.compute_new_clicks(arm_bid)
    margin = np.array([self.env.margin(a) for a in range(self.n_arms_price)])
    crs = self.compute_projection_conversion_rates()
    future_visits = self.compute_future_visits_per_arm()
    cost_per_click = self.compute_cost_per_click(arm_bid)
```

```
    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

def compute_projected_profits_fixed_price(self, arm_price):
    new_clicks = self.compute_projection_new_clicks()
    margin = self.env.margin(arm_price)
    crs = self.compute_conversion_rates(arm_price)
    future_visits = self.compute_future_visits(arm_price)
    cost_per_click = self.compute_cost_per_click_per_arm()

    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
```

In the reported code we can notice some interesting functions that exploit the technique of axis projection. Here are the mentioned functions.

```
def compute_conversion_rates(self, arm_price):
    return sum_ragged_matrix(self.purchases[arm_price]) / \
           sum_ragged_matrix(self.new_clicks[arm_price])

def compute_new_clicks(self, arm_bid):
    return average_ragged_matrix([self.new_clicks[p][arm_bid]
                                  for p in range(self.n_arms_price)])

def compute_cost_per_click(self, arm_bid):
    tot_clicks = np.sum([np.sum(self.new_clicks[arm_p][arm_bid])
                         for arm_p in range(self.n_arms_price)])
    tot_cost = self.tot_cost_per_bid[arm_bid]
    cost_per_click = tot_cost / tot_clicks

    return cost_per_click

def compute_future_visits_per_arm(self):
    return np.array([self.compute_future_visits(arm_p)
                     for arm_p in range(self.n_arms_price)])
```

```
def compute_future_visits(self, arm_price):
    arm_p_future_visits = 0
    arm_p_purchases = 0
    for arm_b in range(self.n_arms_bid):
        complete_samples = len(self.future_visits[arm_price][arm_b])
        future_visits = np.sum(self.future_visits[arm_price][arm_b])
        purchases = np.sum(self.purchases[arm_price][arm_b][:complete_samples])
        arm_p_future_visits += future_visits
        arm_p_purchases += purchases

    return arm_p_future_visits / arm_p_purchases if arm_p_purchases else 0
```

The random variables of this step, namely the conversion rates and the winning probability for an auction, are computed in the same way we did in the previous steps.

```
def compute_projection_conversion_rates(self):
    raise NotImplementedError

def compute_projection_auction_winning_probability(self):
    raise NotImplementedError

def compute_projection_new_clicks(self):
    auction_win_probability = self.compute_projection_auction_winning_probability()
    average_auctions = np.sum(self.tot_auctions_per_bid) / self.current_round

    return auction_win_probability * average_auctions
```

Note the abstract methods that will allow us to implement specific learners to tackle the problem.

### 7.1 The class *UCBOptimalJointLearner*

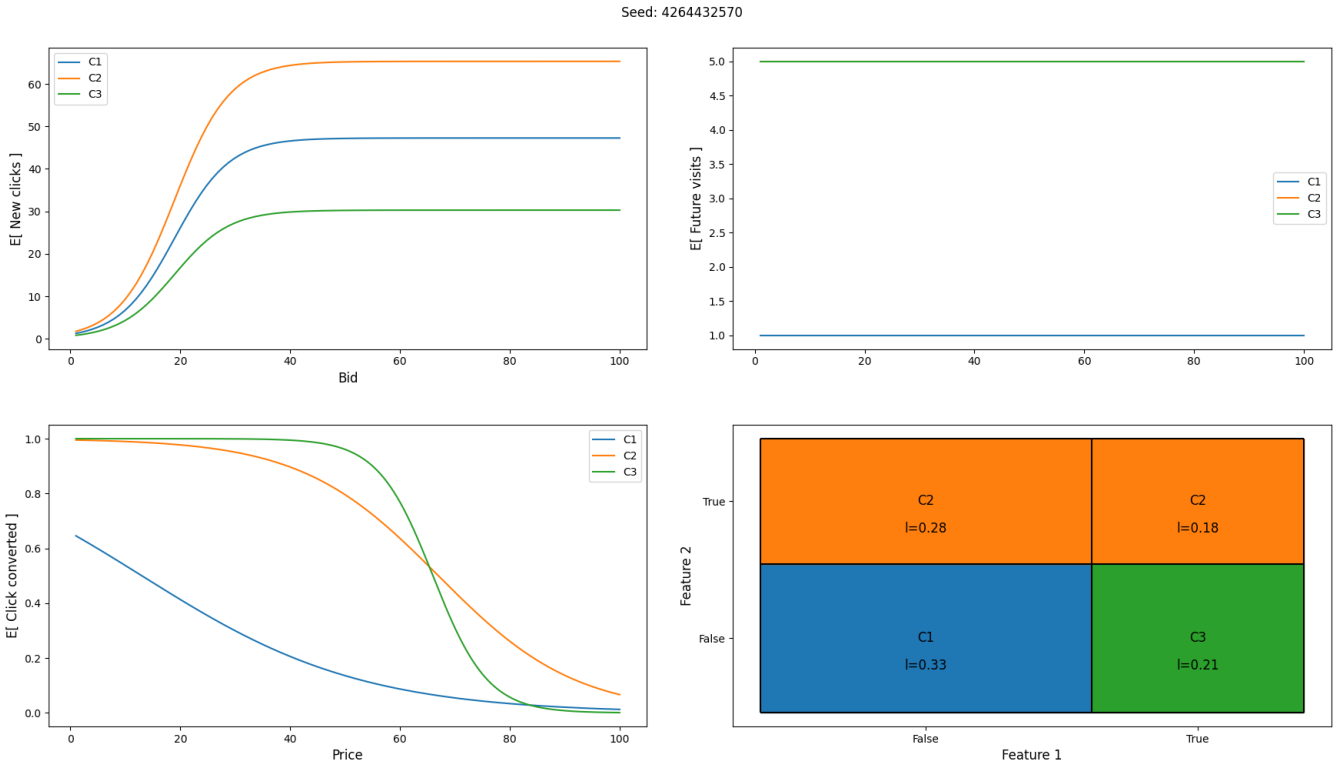
This class implements the abstract methods described before, in order to provide an UCB learner for the given scenario. The most interesting methods are those to compute the radii for the UCB upper bound of our random variables.

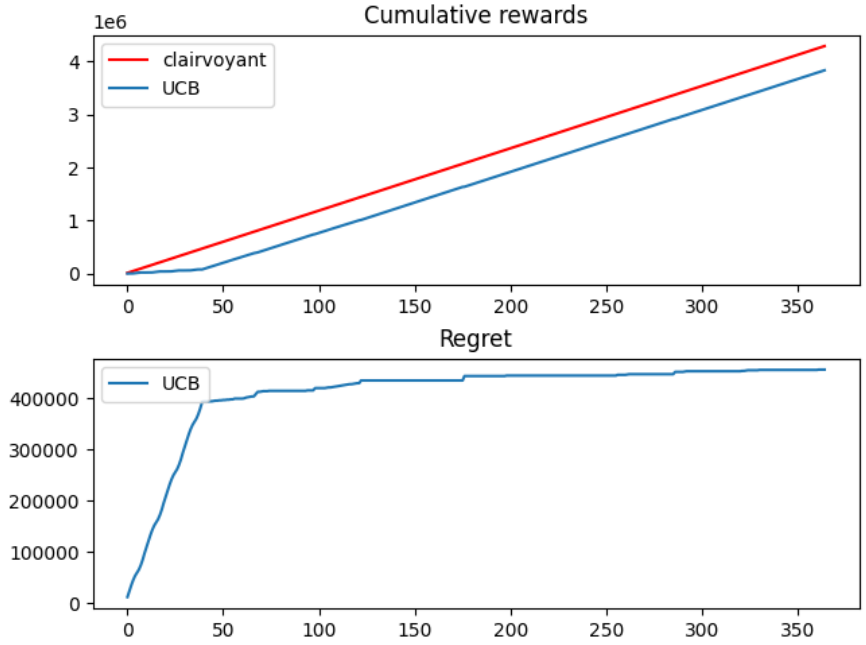
```
def compute_conversion_rates_radii(self):
    tot_clicks_per_arm = np.array([np.sum(sum_ragged_matrix(self.new_clicks[p]))
                                   for p in range(self.n_arms_price)])
    return np.sqrt(2 * np.log(self.current_round) / tot_clicks_per_arm)

def compute_auction_winning_probability_radii(self):
    return (np.sqrt(2 * np.log(self.current_round) /
                    self.tot_auctions_per_bid)).flatten()
```

We need also to compute the average winning probability for each bid. Note that also here we project the data by summing over the price arms.

```
def compute_auction_winning_probability_averages(self):
    new_c = [[] for b in range(self.n_arms_bid)]
    for b in range(self.n_arms_bid):
        for p in range(self.n_arms_price):
            new_c[b].extend(self.new_clicks[p][b])

    return np.array([sum(new_c[b]) / self.tot_auctions_per_bid[b]
                     for b in range(self.n_arms_bid)]).flatten()
```

## 7.2 Experiment 1



Seed: 4264432570
UCB number of pulls:

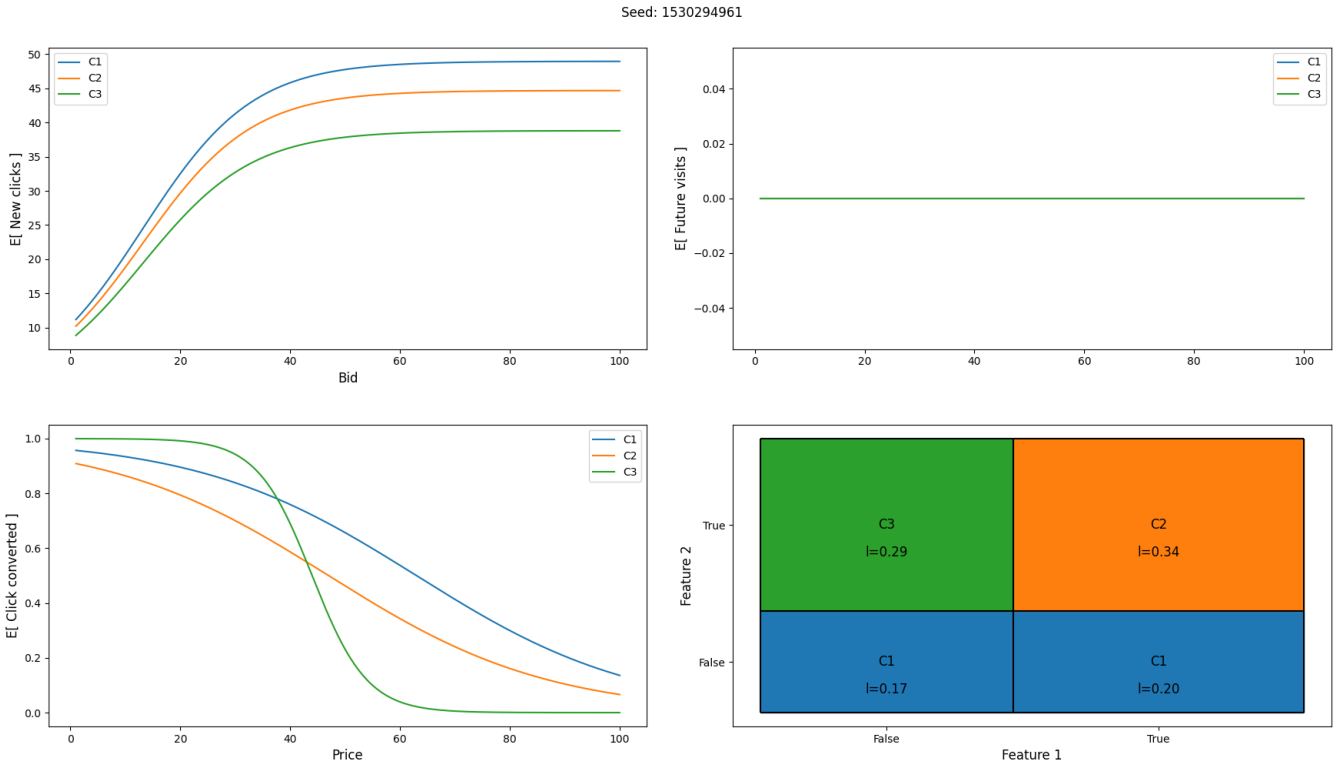
| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20.00 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30.00 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 40.00 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 1 | 0 | 0 |
| 50.00 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 3 | 38 |
| 60.00 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 3 | 275 |
| 70.00 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 4 | 0 |
| 80.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 |
| 90.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| 100.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |

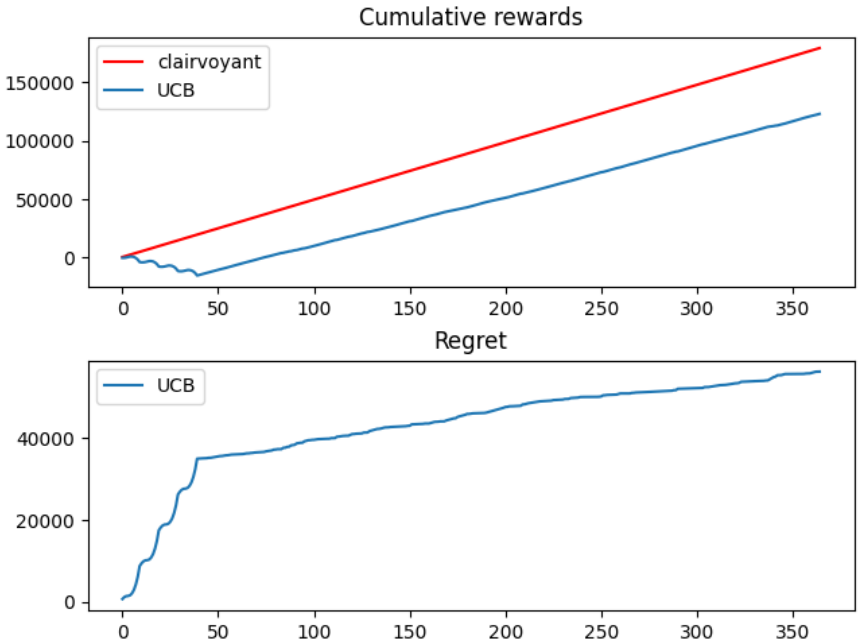
Optimal price: 60.00, Optimal bid: 30.00

Gaps from the optimal arm:

| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 11890 | 12015 | 12242 | 12639 | 13282 | 14595 | 15841 | 17127 | 18286 | 19515 |
| 20.00 | 11724 | 11719 | 11723 | 11754 | 11840 | 12101 | 12430 | 12854 | 13324 | 13953 |
| 30.00 | 11568 | 11441 | 11237 | 10926 | 10490 | 9763 | 9234 | 8849 | 8675 | 8741 |
| 40.00 | 11430 | 11194 | 10804 | 10188 | 9288 | 7684 | 6389 | 5286 | 4538 | 4104 |
| 50.00 | 11327 | 11010 | 10481 | 9638 | 8390 | 6130 | 4265 | 2624 | 1447 | 639 |
| 60.00 | 11307 | 10976 | 10422 | 9536 | 8224 | 5844 | 3872 | 2133 | 877 | 0 |
| 70.00 | 11429 | 11193 | 10801 | 10183 | 9279 | 7669 | 6369 | 5261 | 4509 | 4071 |
| 80.00 | 11563 | 11432 | 11220 | 10897 | 10443 | 9683 | 9123 | 8711 | 8514 | 8561 |
| 90.00 | 11646 | 11579 | 11479 | 11338 | 11161 | 10925 | 10822 | 10840 | 10986 | 11332 |
| 100.00 | 11696 | 11668 | 11635 | 11604 | 11596 | 11677 | 11851 | 12128 | 12482 | 13008 |

## 7.3 Experiment 2



Seed: 1530294961
UCB number of pulls:

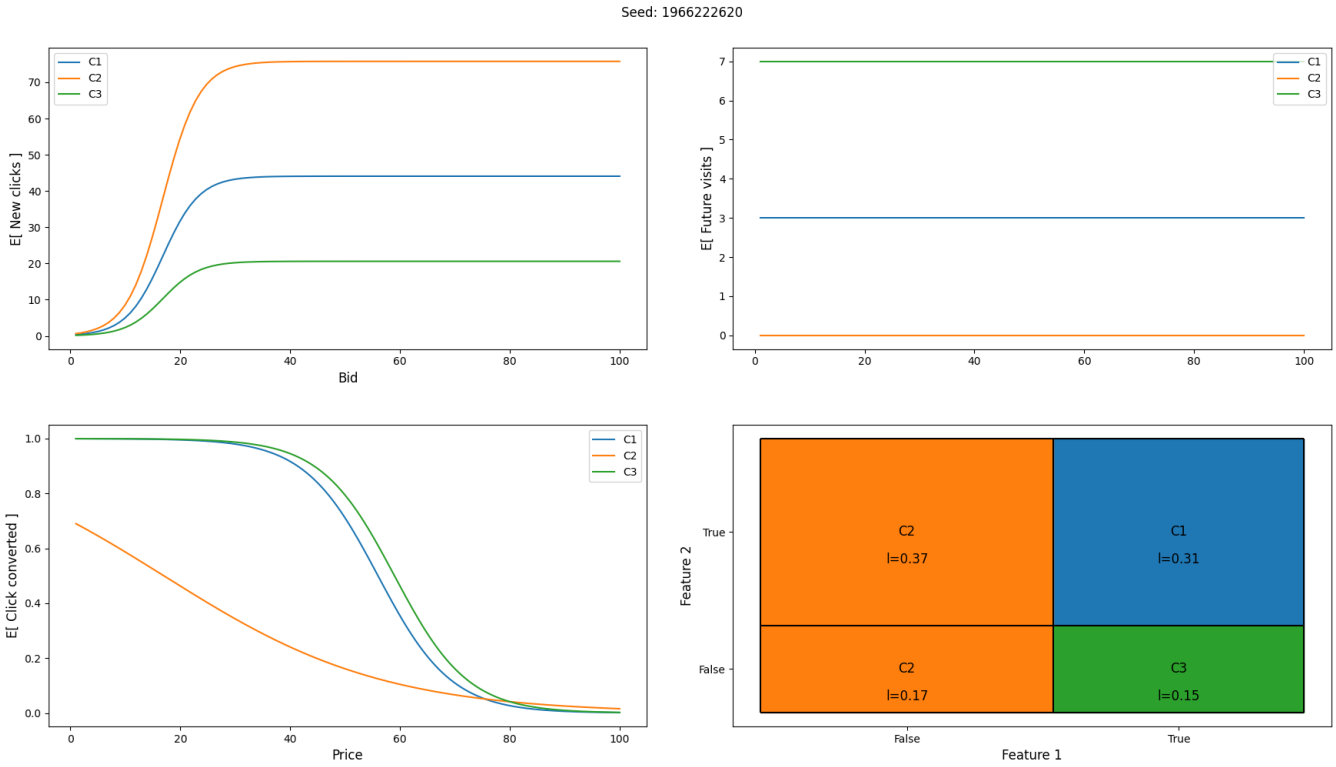
| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20.00 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30.00 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 40.00 | 0 | 28 | 18 | 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| 50.00 | 5 | 69 | 68 | 6 | 4 | 0 | 0 | 0 | 0 | 0 |
| 60.00 | 18 | 38 | 1 | 0 | 0 | 4 | 0 | 0 | 0 | 0 |
| 70.00 | 46 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| 80.00 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| 90.00 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| 100.00 | 11 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |

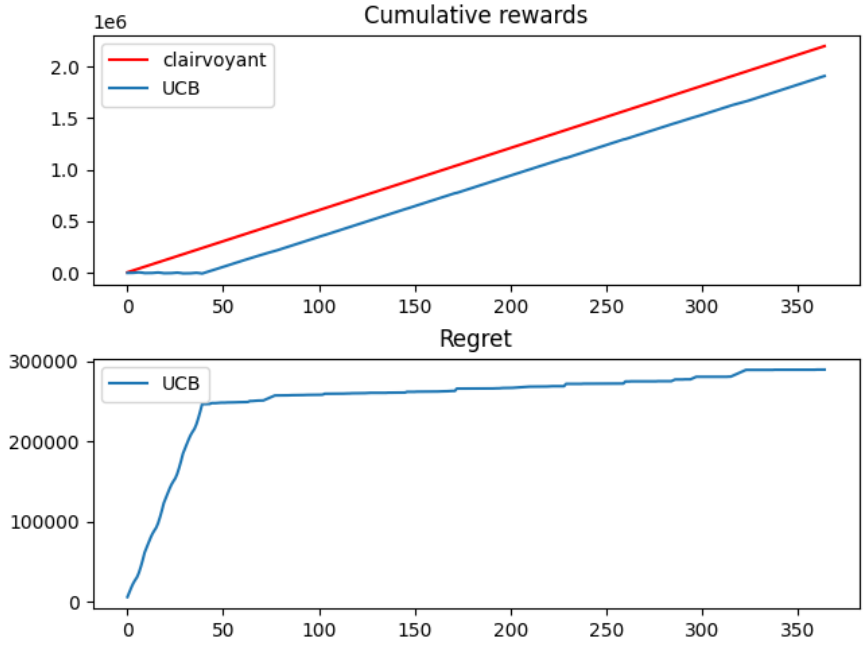
Optimal price: 40.00, Optimal bid: 4.00

Gaps from the optimal arm:

| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 685 | 828 | 1021 | 1269 | 1570 | 2049 | 2452 | 2881 | 3322 | 3914 |
| 20.00 | 409 | 483 | 598 | 759 | 971 | 1329 | 1648 | 2000 | 2375 | 2893 |
| 30.00 | 168 | 182 | 228 | 315 | 448 | 703 | 948 | 1233 | 1550 | 2004 |
| 40.00 | 22 | 0 | 5 | 46 | 132 | 323 | 524 | 769 | 1049 | 1466 |
| 50.00 | 37 | 18 | 26 | 72 | 163 | 360 | 565 | 814 | 1098 | 1518 |
| 60.00 | 83 | 76 | 97 | 158 | 263 | 481 | 700 | 961 | 1257 | 1689 |
| 70.00 | 132 | 137 | 173 | 248 | 369 | 608 | 842 | 1117 | 1425 | 1870 |
| 80.00 | 197 | 218 | 272 | 367 | 510 | 777 | 1031 | 1324 | 1647 | 2109 |
| 90.00 | 267 | 306 | 380 | 498 | 663 | 961 | 1237 | 1549 | 1890 | 2371 |
| 100.00 | 332 | 387 | 480 | 618 | 805 | 1130 | 1426 | 1756 | 2112 | 2611 |

## 7.4 Experiment 3



Seed: 1966222620
UCB number of pulls:

| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20.00 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 30.00 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 40.00 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 21 | 1 | 0 |
| 50.00 | 0 | 0 | 0 | 0 | 4 | 0 | 1 | 33 | 242 | 24 |
| 60.00 | 0 | 0 | 0 | 0 | 0 | 4 | 2 | 1 | 0 | 0 |
| 70.00 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 |
| 80.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| 90.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| 100.00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 |

Optimal price: 50.00, Optimal bid: 26.00

Gaps from the optimal arm:

| p\b | 1.00 | 4.00 | 7.00 | 10.00 | 13.00 | 17.00 | 20.00 | 23.00 | 26.00 | 30.00 |
|---|---|---|---|---|---|---|---|---|---|---|
| 10.00 | 6038 | 6071 | 6157 | 6373 | 6846 | 8015 | 9073 | 9975 | 10645 | 11298 |
| 20.00 | 6006 | 5991 | 5967 | 5938 | 5929 | 6063 | 6329 | 6688 | 7071 | 7568 |
| 30.00 | 5975 | 5917 | 5790 | 5532 | 5073 | 4241 | 3767 | 3620 | 3735 | 4087 |
| 40.00 | 5949 | 5855 | 5642 | 5192 | 4357 | 2716 | 1624 | 1053 | 944 | 1173 |
| 50.00 | 5941 | 5834 | 5592 | 5077 | 4115 | 2200 | 899 | 185 | 0 | 189 |
| 60.00 | 5965 | 5892 | 5732 | 5397 | 4790 | 3637 | 2919 | 2604 | 2630 | 2934 |
| 70.00 | 5998 | 5972 | 5922 | 5833 | 5708 | 5591 | 5666 | 5894 | 6207 | 6667 |
| 80.00 | 6014 | 6011 | 6014 | 6044 | 6153 | 6539 | 6998 | 7490 | 7943 | 8478 |
| 90.00 | 6019 | 6023 | 6044 | 6113 | 6299 | 6850 | 7436 | 8014 | 8513 | 9073 |
| 100.00 | 6020 | 6027 | 6053 | 6135 | 6345 | 6948 | 7574 | 8179 | 8692 | 9260 |

# 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.

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

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

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

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

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

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

therefore

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

And by definition of the likelihood $l_c$ of a class:

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

## 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.