Advertising and Pricing

Gabriele Daglio, Federico Di Cesare, Jacopo Germano

July 14, 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 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 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 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:

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

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 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:

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

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 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*.

```python
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 $(C_{c,b,i})_{i=1,\ldots,N_{c,b}}$ of random variables representing the cost paid for each click i, such that $C_{c,b,i} \sim CostPerClick(c,b)$

- The sequence $(D_{c,p,i})_{i=1,\ldots,N_{c,b}}$ 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 $(F_{c,i})_{i=1,\ldots,N_{c,b}}$ 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:

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

And therefore formulate the optimization problem as follows:

$$\underset{p,b}{\arg\max} \; ExpectedProfit(p,b)$$

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:

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

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:

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)*:

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

    combs = combinations if combinations is not None else env.combinations

    profit = sum(
        simple_class_profit(margin=m(p),
                            new_clicks=n[comb],
                            conversion_rate=r(env.class_of_comb[comb], p),
                            future_visits=f(env.class_of_comb[comb]),
                            cost_per_click=k(env.class_of_comb[comb], b))
        for comb in combs
    )

    return profit
```

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

```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)
```

The algorithm is as follows:

```python
def step1(env, prices, bids, combinations=None):
    median_b = bids[len(bids) // 2]

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

    profit = expected_profit(env, optimal_price, optimal_bid, combinations)

    return optimal_price, optimal_bid, profit
```

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

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

    return prices[opt_p_index]


def optimal_bid_for_price(env, bids, price, combinations=None):
    opt_b_index = np.argmax([
        expected_profit(env, price, b, combinations=combinations)
        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 $\bar{a}$ as the sample mean of the previously observed $a_j s$:

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

- 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. 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.*

Note: *The future visits per purchase $\overline{f_p}$ can be estimated independently of the bid value that was used: in fact, the observed future visits per purchase depend on the proportion of each class with respect to the total number of purchases with a price. Each price p will induce a different proportion in the classes of users that have purchased the product, due to different reserve prices: if before the purchase the weights x_c of proportion are the class likelihoods $x_c = l_c$, and the percentage of each class c is $\frac{x_c}{\sum_j x_j}$, after the purchase the weights are influenced by the conversion rate: $x_{c,p} = l_c r(c,p)$. Given a price, any change in the bid will preserve, on average, the proportion induced by that price: therefore every different purchase on a round where the strategy (p,b) was used counts as a sample of the future visits associated with price p, and the only difference between different bid values is that the highest bids will provide more samples.*

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:

```python
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:

```python
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.

14

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:

```python
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.

```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
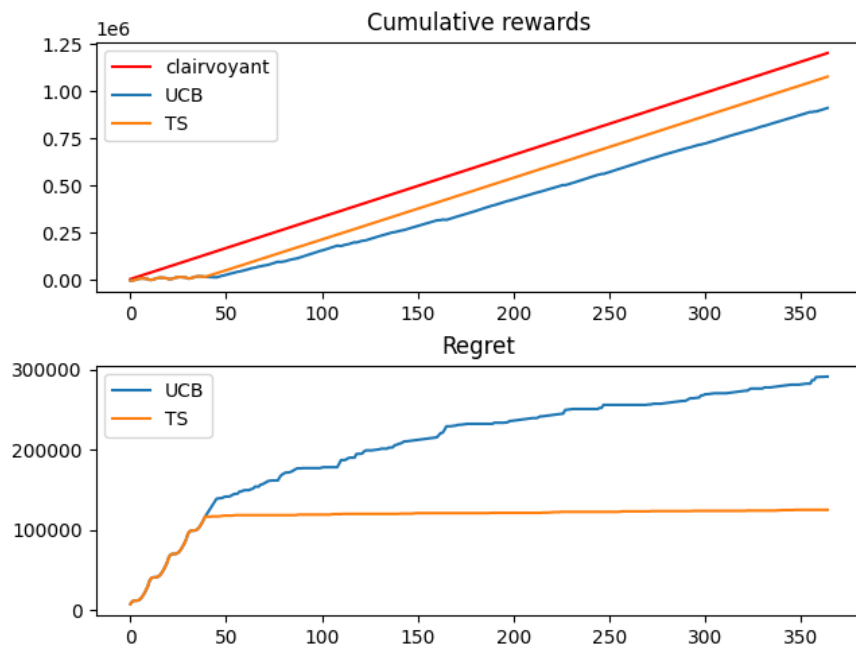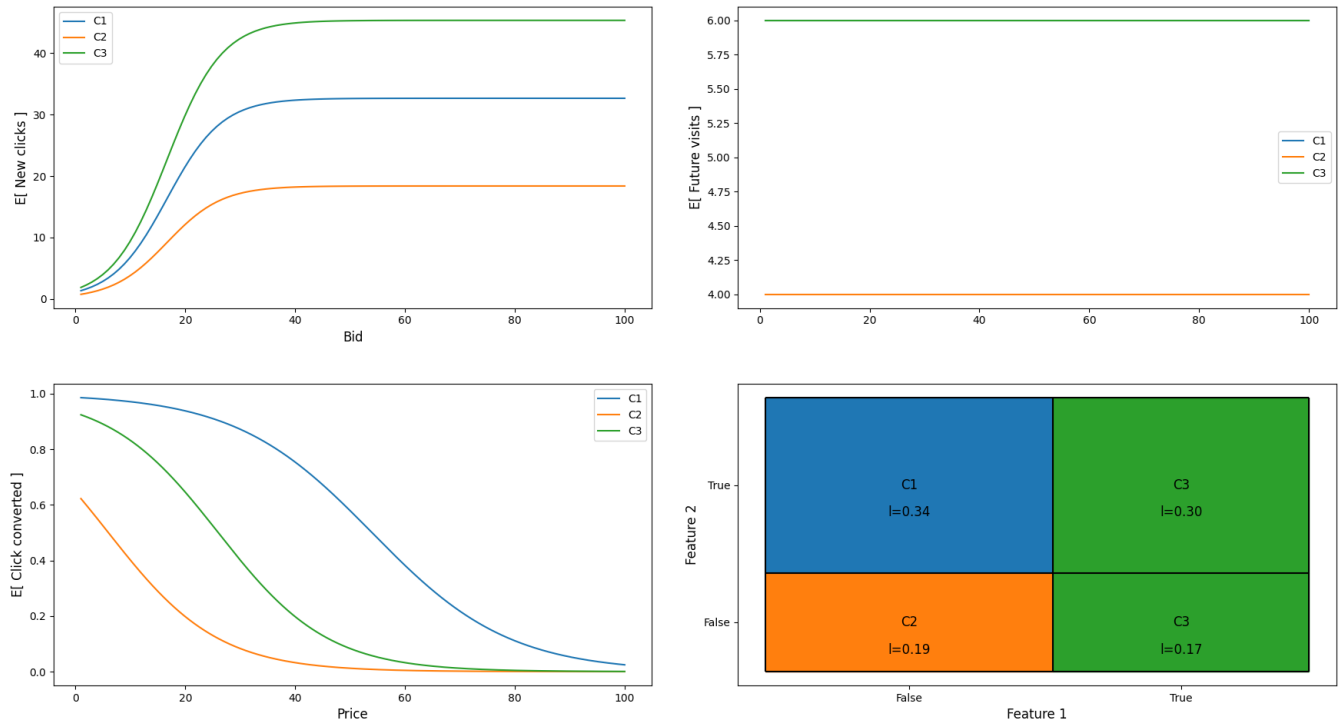
Seed: 3144548588

Price	Expected	Norm. gaps	UCB Pulls	UCB Expected	TS Pulls	TS Expected
10.00	-4309.67	100.0	4	-4287.03	4	-4447.41
20.00	-48.64	43.9	4	16.98	4	28.71
30.00	2460.29	10.9	10	2650.07	4	2066.38
40.00	3291.72	0.0	141	3266.95	302	3344.78
50.00	2977.05	4.1	131	3045.23	31	3016.02
60.00	1924.81	18.0	34	2020.80	4	1859.43
70.00	632.38	35.0	13	218.62	4	-180.93
80.00	-440.08	49.1	13	-435.83	4	-165.58
90.00	-1144.79	58.4	11	-1236.55	4	-1492.98
100.00	-1550.37	63.7	4	-1806.97	4	-1715.24

Optimal price: 40.00, Optimal bid: 27.00

4.4   Experiment 2
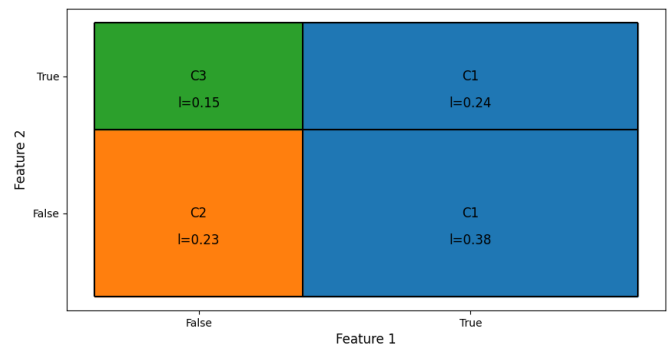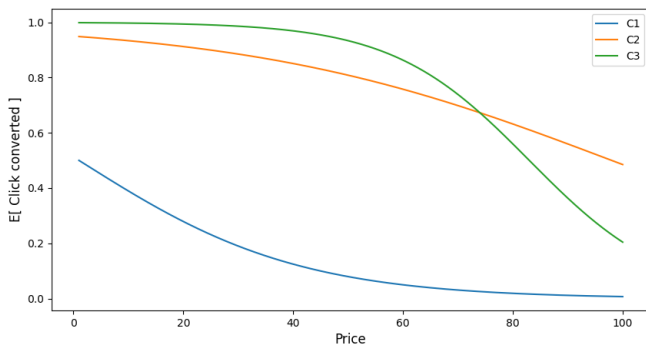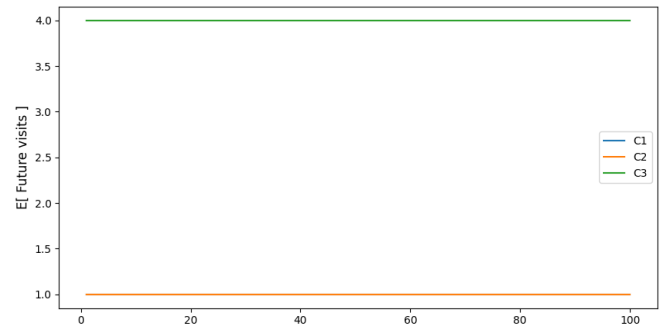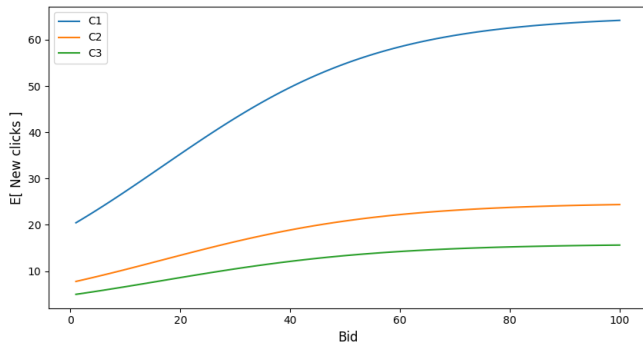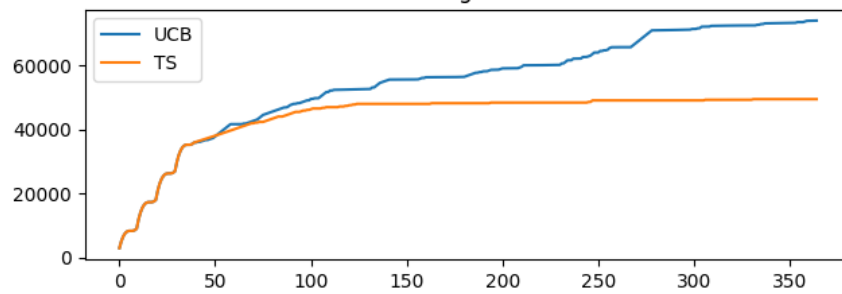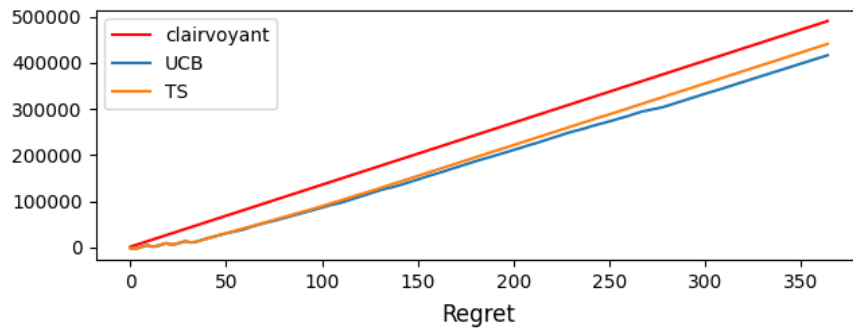
Seed: 1873674269

Price	Expected	Norm. gaps	UCB Pulls	UCB Expected	TS Pulls	TS Expected
10.00	-1564.32	100.0	4	-1677.11	4	-1704.78
20.00	-806.12	73.9	4	-775.53	4	-778.56
30.00	-172.82	52.1	4	-200.15	4	-153.46
40.00	361.10	33.8	4	264.64	4	414.62
50.00	808.49	18.4	5	765.44	4	993.70
60.00	1152.60	6.5	14	1044.73	27	1177.04
70.00	1342.60	0.0	95	1351.48	261	1351.32
80.00	1327.18	0.5	122	1336.60	7	825.18
90.00	1131.00	7.3	65	1138.87	45	1097.34
100.00	865.62	16.4	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:

```python
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.

```python
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.

```python
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()*:

```python
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:

```python
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:

```python
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:

```python
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()*:

```python
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
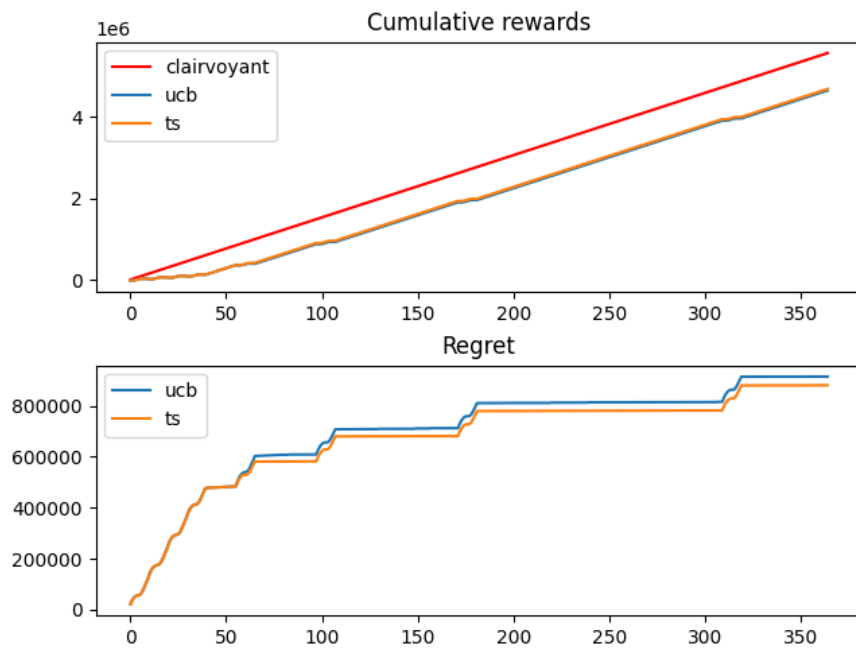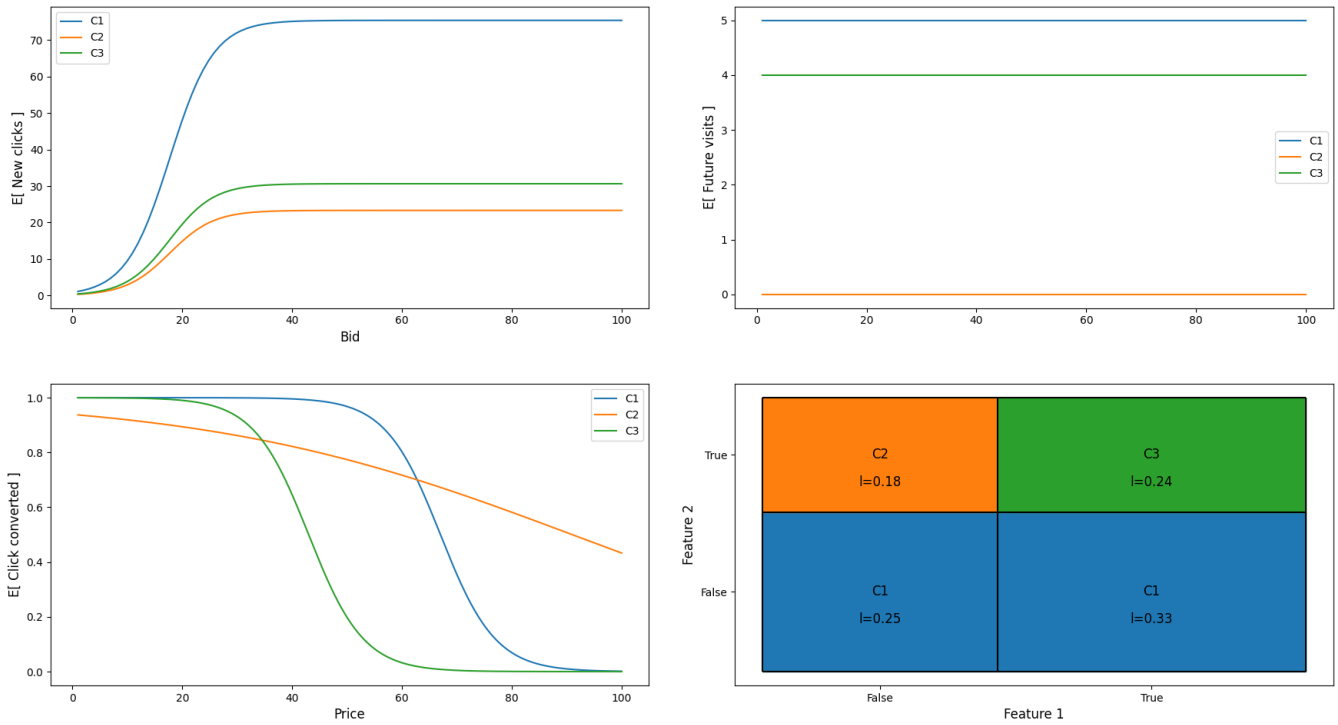
```
Seed: 3511939391
Legend: context, normalized gap, number of pulls
UCB with context generation:
+--------+-------+-------+-------+----+-------+-------+----+-------+-------+
| Price  | TF,FF | Gaps  | Pulls | TT | Gaps  | Pulls | FT | Gaps  | Pulls |
+--------+-------+-------+-------+----+-------+-------+----+-------+-------+
|  10.00 |       | 100.0 |     8 |    | 100.0 |     5 |    | 100.0 |     5 |
|  20.00 |       |  75.8 |     8 |    |  53.7 |     5 |    |  78.9 |     5 |
|  30.00 |       |  51.6 |     8 |    |  11.5 |     8 |    |  59.2 |     5 |
|  40.00 |       |  27.6 |     8 |    |   0.0 |   315 |    |  41.4 |    32 |
|  50.00 |       |   5.8 |    14 |    |  40.8 |     6 |    |  26.1 |     5 |
|  60.00 |       |   0.0 |   287 |    |  65.5 |     6 |    |  13.9 |     8 |
|  70.00 |       |  39.1 |     8 |    |  71.0 |     5 |    |   5.4 |    42 |
|  80.00 |       |  74.8 |     8 |    |  71.9 |     5 |    |   0.7 |    43 |
|  90.00 |       |  83.7 |     8 |    |  72.1 |     5 |    |   0.0 |   143 |
| 100.00 |       |  85.2 |     8 |    |  72.1 |     5 |    |   2.8 |    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 | Gaps  | Pulls | TT | Gaps  | Pulls | FT | Gaps  | Pulls |
+--------+-------+-------+-------+----+-------+-------+----+-------+-------+
|  10.00 |       | 100.0 |     8 |    | 100.0 |     4 |    | 100.0 |     4 |
|  20.00 |       |  75.8 |     8 |    |  53.7 |     4 |    |  78.9 |     4 |
|  30.00 |       |  51.6 |     8 |    |  11.5 |     4 |    |  59.2 |     4 |
|  40.00 |       |  27.6 |     8 |    |   0.0 |   328 |    |  41.4 |    17 |
|  50.00 |       |   5.8 |     8 |    |  40.8 |     4 |    |  26.1 |     4 |
|  60.00 |       |   0.0 |   293 |    |  65.5 |     5 |    |  13.9 |     5 |
|  70.00 |       |  39.1 |     8 |    |  71.0 |     4 |    |   5.4 |    17 |
|  80.00 |       |  74.8 |     8 |    |  71.9 |     4 |    |   0.7 |    99 |
|  90.00 |       |  83.7 |     8 |    |  72.1 |     4 |    |   0.0 |    53 |
| 100.00 |       |  85.2 |     8 |    |  72.1 |     4 |    |   2.8 |   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
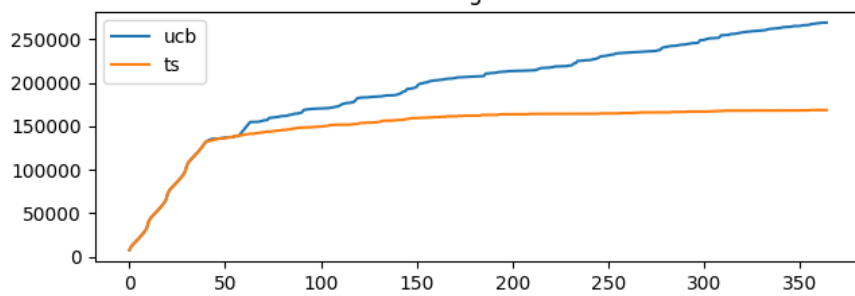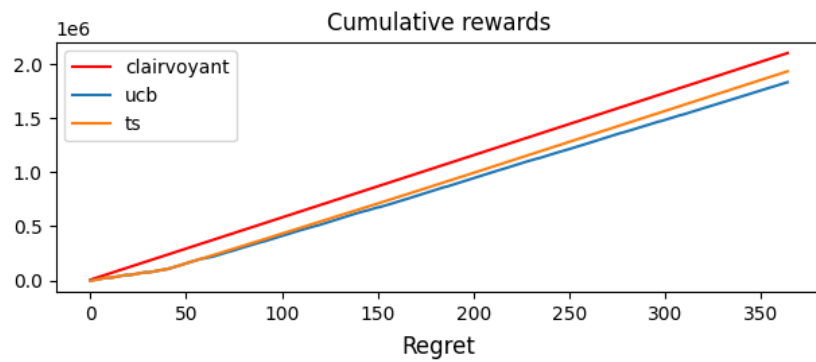
```
Seed: 1740212098
Legend: context, normalized gap, number of pulls
UCB with context generation:
```

Price	TF	Gaps	Pulls	FF	Gaps	Pulls	TT	Gaps	Pulls	FT	Gaps	Pulls
10.00		100.0	4		100.0	4		100.0	4		100.0	4
20.00		32.1	4		0.0	4		0.0	6		71.3	4
30.00		0.0	297		43.7	4		43.7	12		46.5	4
40.00		29.3	17		58.6	7		58.6	10		26.5	4
50.00		68.2	6		61.8	125		61.8	20		12.0	12
60.00		85.3	7		62.3	23		62.3	30		3.2	30
70.00		90.6	4		62.4	31		62.4	44		0.0	91
80.00		92.1	18		62.5	43		62.5	59		1.7	89
90.00		92.5	4		62.5	54		62.5	79		7.3	100
100.00		92.6	4		62.5	70		62.5	101		15.5	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	Gaps	Pulls	FT	Gaps	Pulls	TF	Gaps	Pulls	FF	Gaps	Pulls
10.00		100.0	4		100.0	4		100.0	4		100.0	4
20.00		0.0	263		71.3	4		32.1	4		0.0	262
30.00		43.7	11		46.5	4		0.0	328		43.7	27
40.00		58.6	19		26.5	4		29.3	4		58.6	5
50.00		61.8	8		12.0	15		68.2	4		61.8	10
60.00		62.3	11		3.2	11		85.3	5		62.3	7
70.00		62.4	11		0.0	245		90.6	4		62.4	10
80.00		62.5	10		1.7	6		92.1	4		62.5	13
90.00		62.5	13		7.3	60		92.5	4		62.5	13
100.00		62.5	15		15.5	12		92.6	4		62.5	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
```
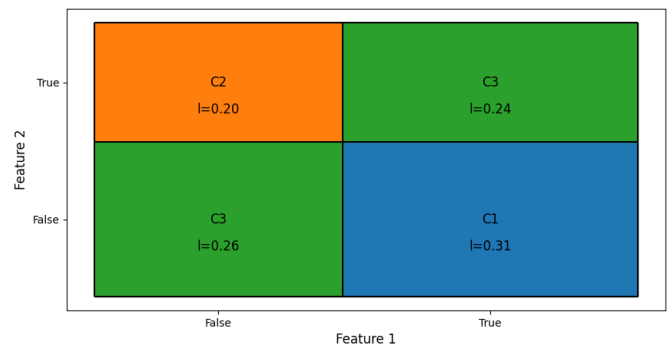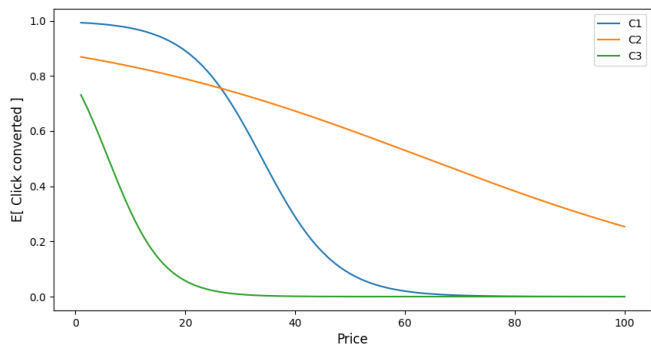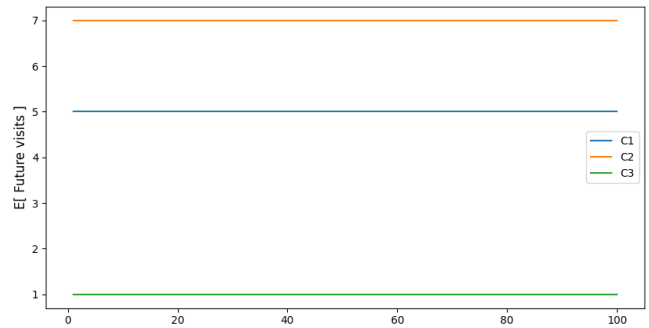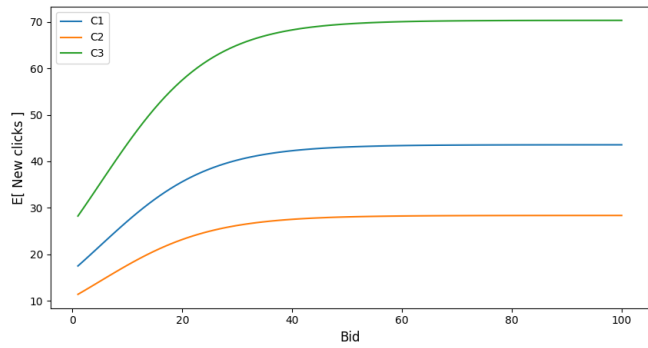
5.6 Experiment 3
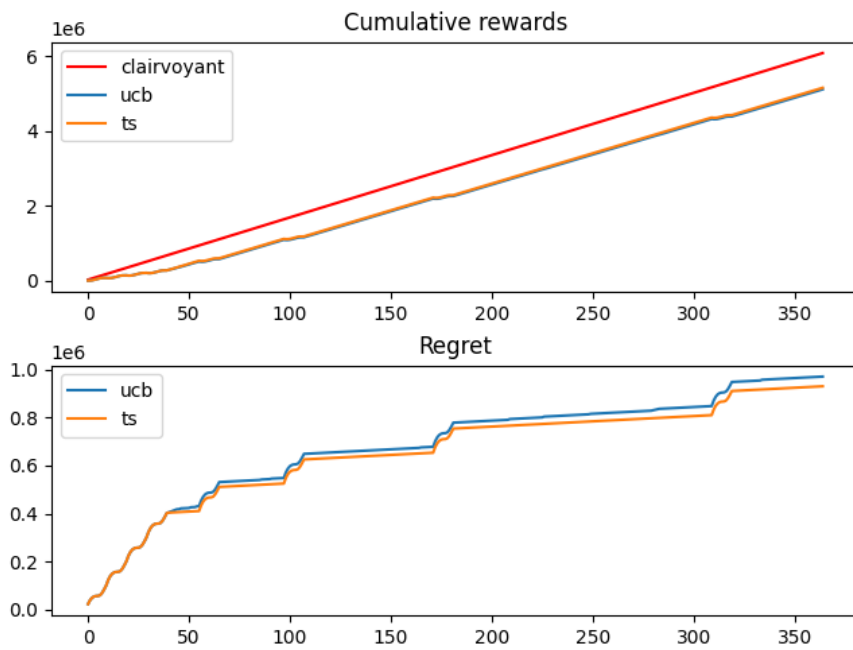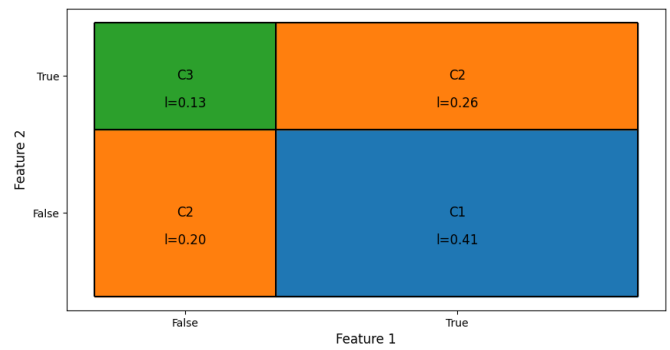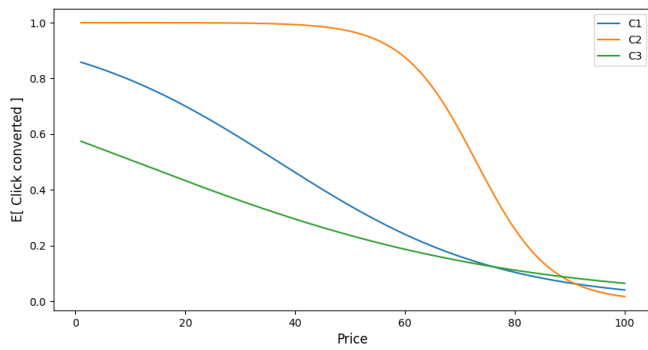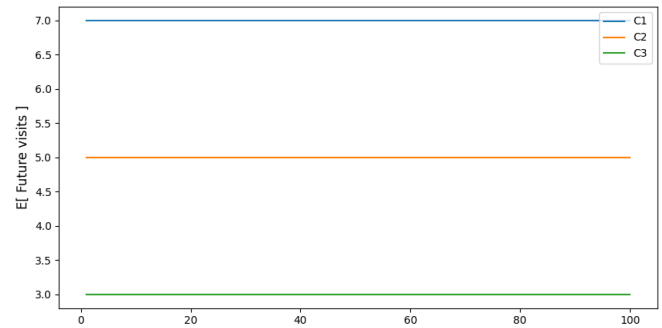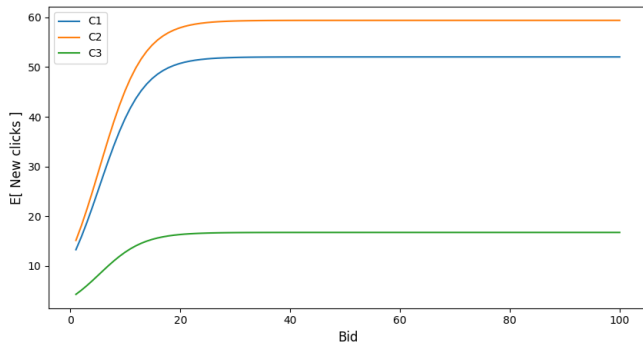
```
Seed: 4059059292
Legend: context, normalized gap, number of pulls
UCB with context generation:
+--------+-------------+-------+-------+
| Price  | TT,TF,FT,FF |  Gaps | Pulls |
+--------+-------------+-------+-------+
|  10.00 |             | 100.0 |     8 |
|  20.00 |             |  69.4 |     8 |
|  30.00 |             |  43.2 |     8 |
|  40.00 |             |  22.1 |     8 |
|  50.00 |             |   6.7 |    16 |
|  60.00 |             |   0.0 |   274 |
|  70.00 |             |  12.1 |    19 |
|  80.00 |             |  41.3 |     8 |
|  90.00 |             |  62.2 |     8 |
| 100.00 |             |  71.1 |     8 |
+--------+-------------+-------+-------+

 Performed splits:

 TS with context generation:
+--------+-------------+-------+-------+
| Price  | TT,TF,FT,FF |  Gaps | Pulls |
+--------+-------------+-------+-------+
|  10.00 |             | 100.0 |     8 |
|  20.00 |             |  69.4 |     8 |
|  30.00 |             |  43.2 |     8 |
|  40.00 |             |  22.1 |     8 |
|  50.00 |             |   6.7 |     9 |
|  60.00 |             |   0.0 |   292 |
|  70.00 |             |  12.1 |     8 |
|  80.00 |             |  41.3 |     8 |
|  90.00 |             |  62.2 |     8 |
| 100.00 |             |  71.1 |     8 |
+--------+-------------+-------+-------+

 Performed splits:


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

5.7   Final considerations about UCB vs TS

We found some discrepancies between the behaviour of Thompson Sampling and Upper Confidence Bound, leading to potential differences in the early convergence. We are estimating a conversion rate, which is lower as the price goes up and higher as the price approaches zero. Due to the nature of the problem, a small change in the conversion rate can lead to dramatic changes in the expected profit.

What might happen is that given a high price, the real conversion rate is very low and so is the expected profit. Trivially if an item is very expensive only a few people will buy it, then the company producing that item will earn nothing or even lose money.

What happens if for some reason a small amount of people start buying that product? Suddenly, due to the high price, the revenue of the company skyrockets! This is exactly what happens if we overestimate the conversion rates for arms with a high price: the expected profit overshoots.

If we think about the mechanisms behind UCB and TS we can clearly see that UCB tends to be optimistic and will always consider a higher value for the random variable it's estimating. On the other hand TS works by drawing samples from a prior distribution thus on the long run it won't be optimistic nor pessimistic, it will just consider the random variables with some noise.

What are the consequences on the expected profit? UCB will give more shots to the higher prices since as we said before it overestimates the conversion rates and a small change in it will result in a much higher profit.

For this reason, in some experiments TS converges faster, while UCB "wastes time" exploring very expensive arms, but due to the nature of the confidence bounds, the regret will still be logarithmic and also UCB converges in a decent number of rounds.

From now on, we will use only a UCB-like approach. Despite the disadvantages we have just discussed, we think that a deterministic algorithm allows us to study in more depth the problem since we will face only decisions caused by math and we won't have randomness in the experiments (also facilitating the reproducibility, core aspect of the scientific method).

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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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
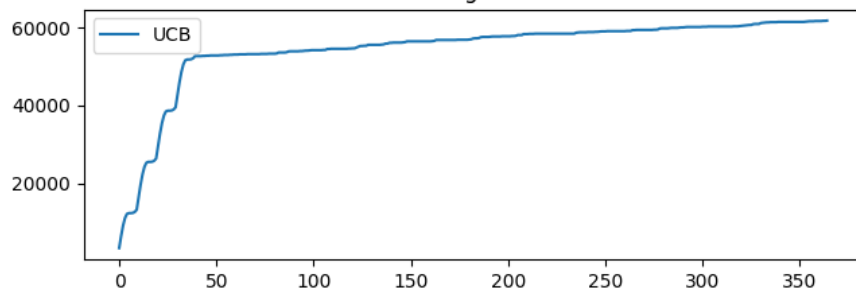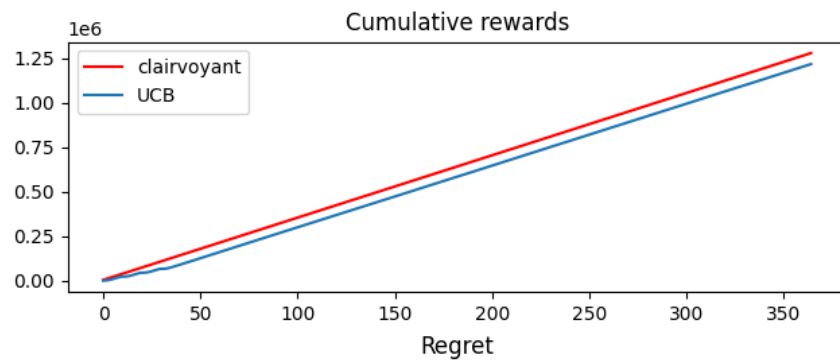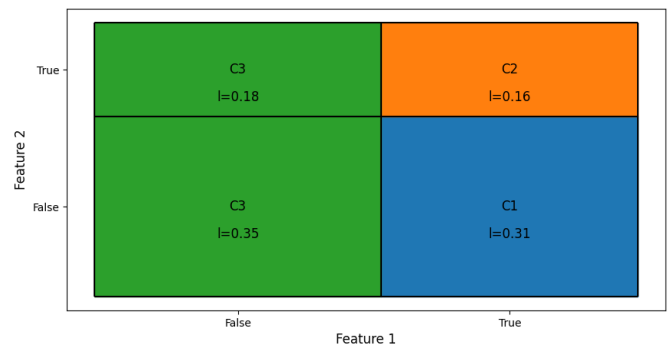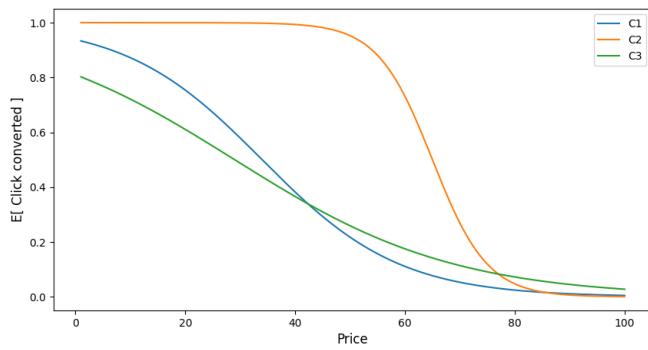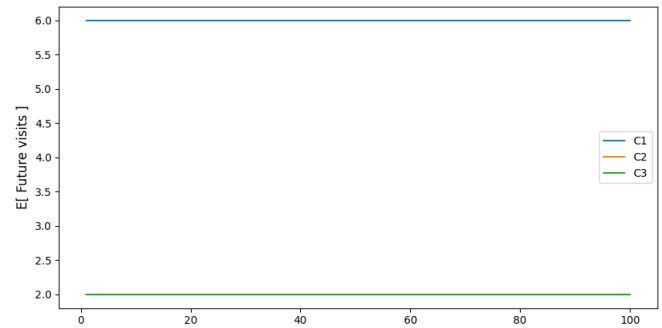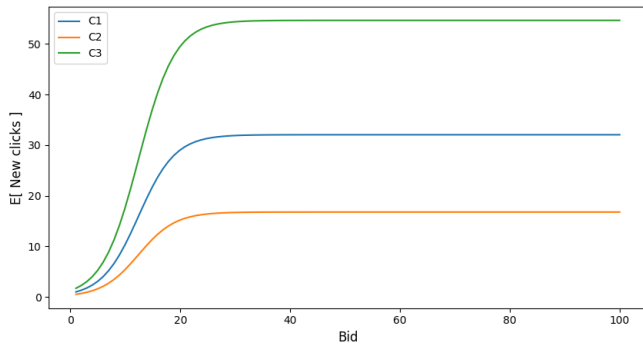
```
Seed: 3077083550
+-------+--------------+-------+-------+-----------------+
|   Bid | True expected |  Gaps | Pulls | Learner expected |
+-------+--------------+-------+-------+-----------------+
|  1.00 |       176.27 | 100.0 |     4 |          221.40 |
|  4.00 |       394.24 |  93.4 |     4 |          365.96 |
|  7.00 |       830.11 |  80.4 |     4 |          860.37 |
| 10.00 |      1557.46 |  58.5 |     4 |         1601.24 |
| 13.00 |      2446.80 |  31.8 |     4 |         2370.49 |
| 17.00 |      3302.31 |   6.1 |    15 |         3224.52 |
| 20.00 |      3503.98 |   0.0 |   226 |         3506.75 |
| 23.00 |      3446.78 |   1.7 |    87 |         3462.11 |
| 26.00 |      3261.94 |   7.3 |    13 |         3279.35 |
| 30.00 |      2937.62 |  17.0 |     4 |         2931.16 |
+-------+--------------+-------+-------+-----------------+

Optimal price: 42.00, Optimal bid: 20.00
```

6.3 Experiment 2
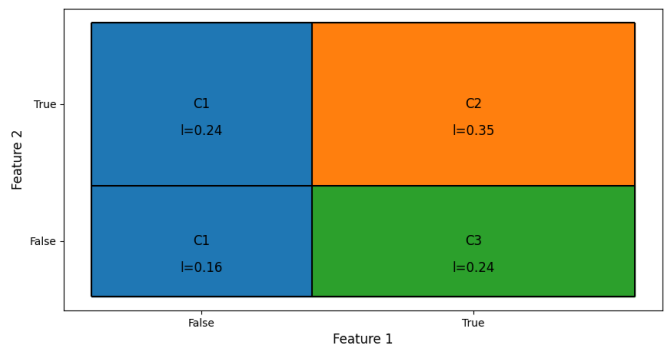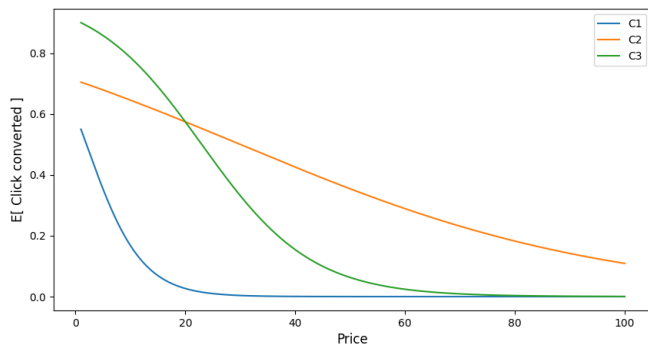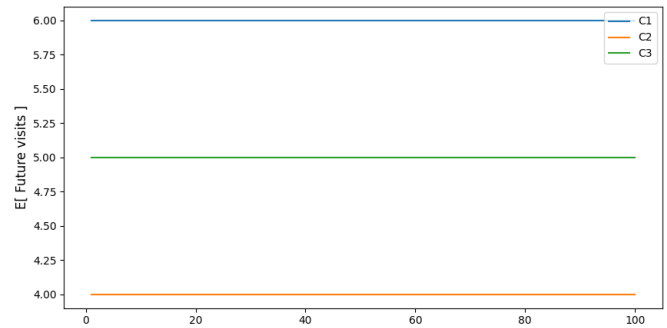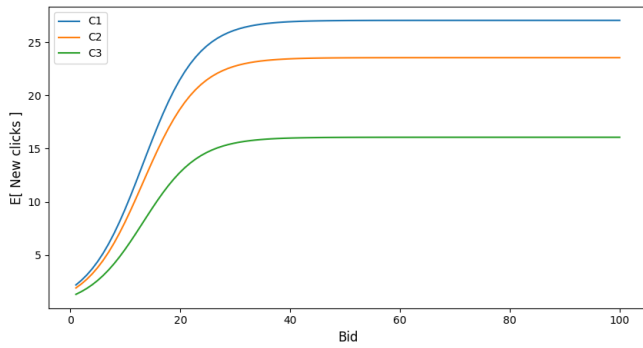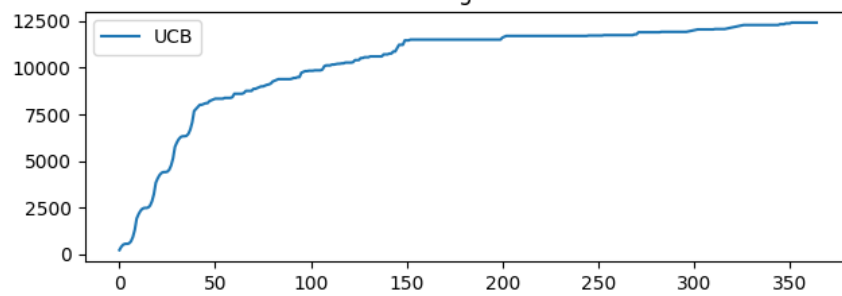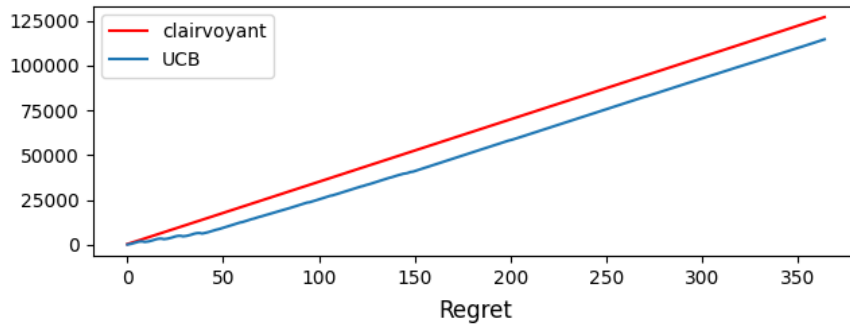
```
Seed: 538541279
+-------+--------------+-------+-------+----------------+
|   Bid | True expected |  Gaps | Pulls | Learner expected |
+-------+--------------+-------+-------+----------------+
|  1.00 |       112.66 |  38.6 |     6 |          92.84 |
|  4.00 |       168.78 |  29.4 |     8 |         162.38 |
|  7.00 |       237.41 |  18.2 |    17 |         229.92 |
| 10.00 |       305.21 |   7.0 |    39 |         281.62 |
| 13.00 |       348.09 |   0.0 |   247 |         322.53 |
| 17.00 |       326.47 |   3.5 |    32 |         304.58 |
| 20.00 |       243.51 |  17.2 |     4 |         192.83 |
| 23.00 |       116.61 |  38.0 |     4 |          75.33 |
| 26.00 |       -36.98 |  63.2 |     4 |         -70.67 |
| 30.00 |      -261.17 | 100.0 |     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 the next arm. Here we exploit the results from Step 1, where we stated that the optimal price is the same 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.

```python
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(arm_bid)* and *compute_projected_profits_fixed_price(arm_price)* which are reported below.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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
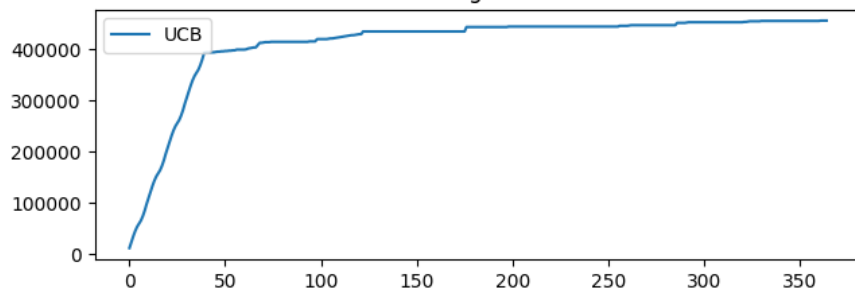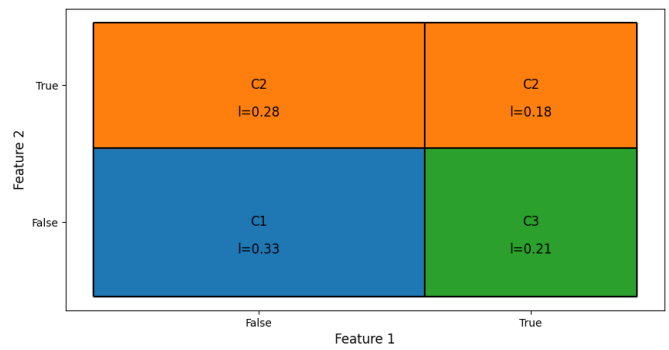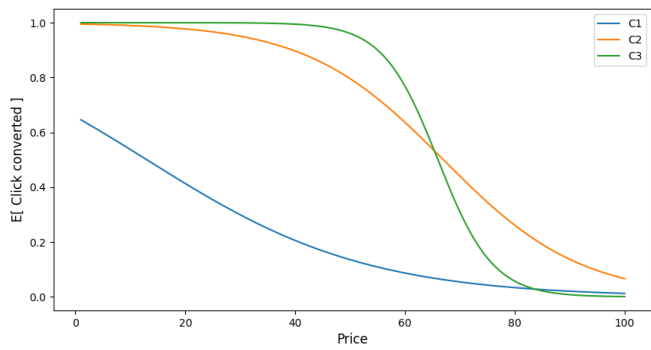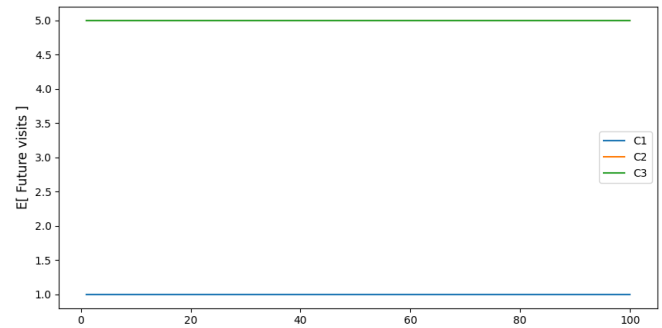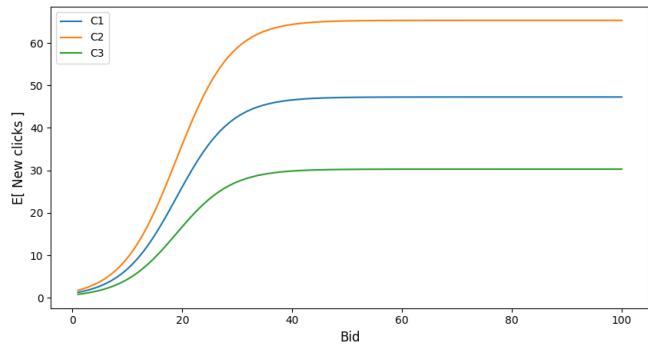
Seed: 4264432570
UCB number of pulls:

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

Rescaled 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	60.9	61.6	62.7	64.8	68.1	74.8	81.2	87.8	93.7	100.0
20.00	60.1	60.1	60.1	60.2	60.7	62.0	63.7	65.9	68.3	71.5
30.00	59.3	58.6	57.6	56.0	53.8	50.0	47.3	45.3	44.5	44.8
40.00	58.6	57.4	55.4	52.2	47.6	39.4	32.7	27.1	23.3	21.0
50.00	58.0	56.4	53.7	49.4	43.0	31.4	21.9	13.4	7.4	3.3
60.00	57.9	56.2	53.4	48.9	42.1	29.9	19.8	10.9	4.5	0.0
70.00	58.6	57.4	55.3	52.2	47.5	39.3	32.6	27.0	23.1	20.9
80.00	59.3	58.6	57.5	55.8	53.5	49.6	46.7	44.6	43.6	43.9
90.00	59.7	59.3	58.8	58.1	57.2	56.0	55.5	55.5	56.3	58.1
100.00	59.9	59.8	59.6	59.5	59.4	59.8	60.7	62.1	64.0	66.7

7.3   Experiment 2
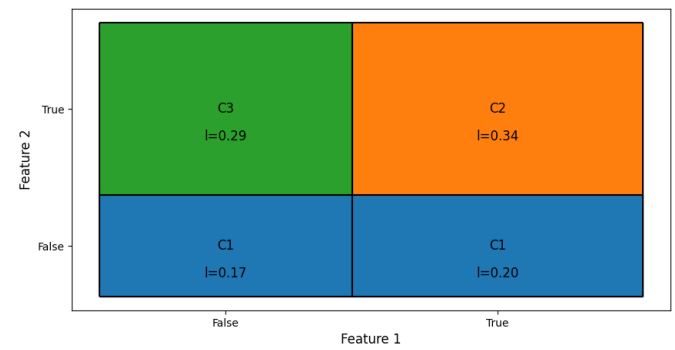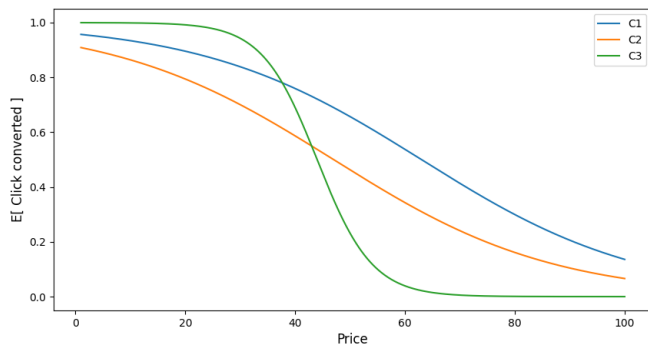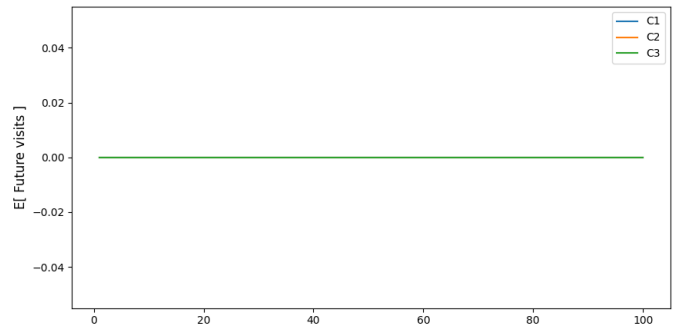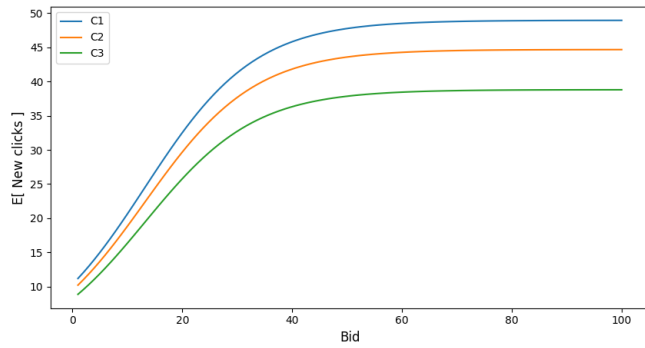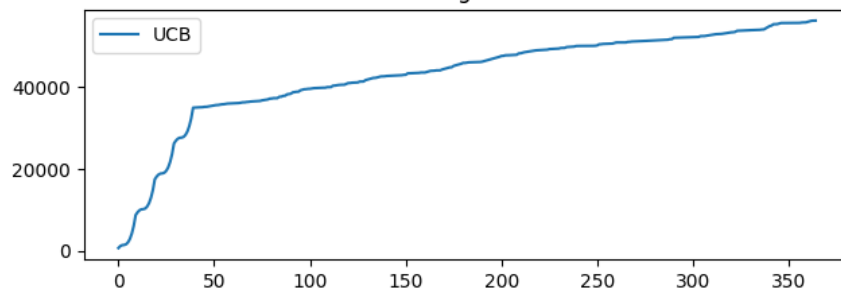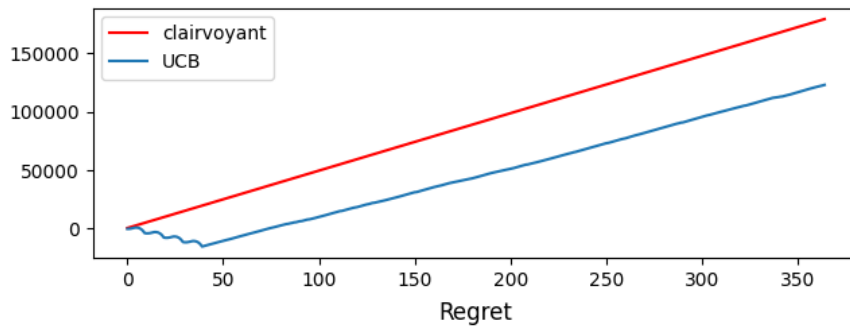
```
Seed: 1530294961
UCB number of pulls:
```

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

Rescaled 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	17.5	21.2	26.1	32.4	40.1	52.3	62.7	73.6	84.9	100.0
20.00	10.5	12.3	15.3	19.4	24.8	34.0	42.1	51.1	60.7	73.9
30.00	4.3	4.7	5.8	8.0	11.5	18.0	24.2	31.5	39.6	51.2
40.00	0.6	0.0	0.1	1.2	3.4	8.3	13.4	19.6	26.8	37.5
50.00	0.9	0.5	0.7	1.8	4.2	9.2	14.4	20.8	28.1	38.8
60.00	2.1	1.9	2.5	4.0	6.7	12.3	17.9	24.6	32.1	43.2
70.00	3.4	3.5	4.4	6.3	9.4	15.5	21.5	28.6	36.4	47.8
80.00	5.0	5.6	6.9	9.4	13.0	19.8	26.3	33.8	42.1	53.9
90.00	6.8	7.8	9.7	12.7	17.0	24.6	31.6	39.6	48.3	60.6
100.00	8.5	9.9	12.3	15.8	20.6	28.9	36.4	44.9	54.0	66.7

7.4   Experiment 3
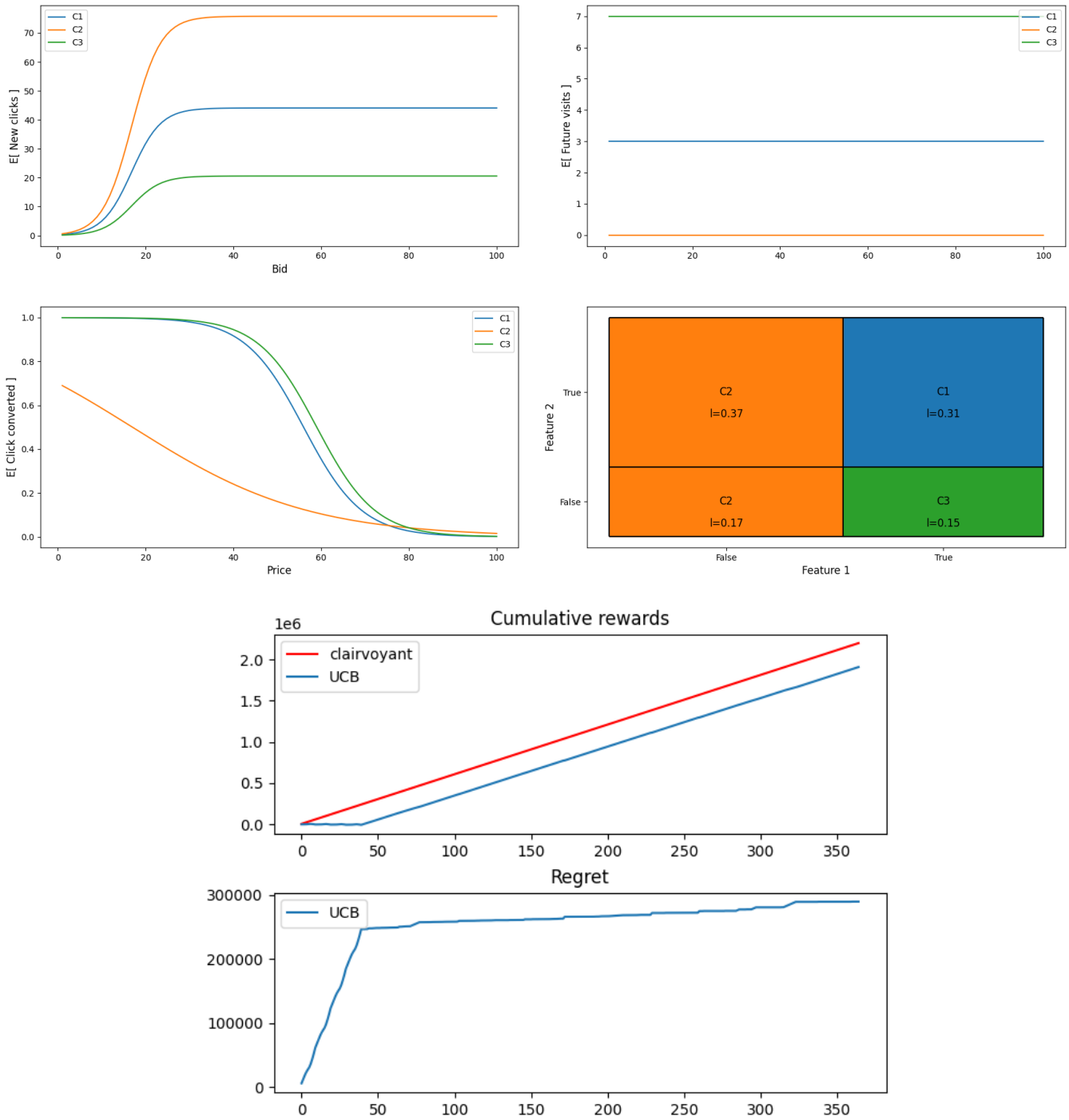
Seed: 1966222620
UCB number of pulls:

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

Rescaled 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	53.4	53.7	54.5	56.4	60.6	70.9	80.3	88.3	94.2	100.0
20.00	53.2	53.0	52.8	52.6	52.5	53.7	56.0	59.2	62.6	67.0
30.00	52.9	52.4	51.2	49.0	44.9	37.5	33.3	32.0	33.1	36.2
40.00	52.7	51.8	49.9	46.0	38.6	24.0	14.4	9.3	8.4	10.4
50.00	52.6	51.6	49.5	44.9	36.4	19.5	8.0	1.6	0.0	1.7
60.00	52.8	52.2	50.7	47.8	42.4	32.2	25.8	23.1	23.3	26.0
70.00	53.1	52.9	52.4	51.6	50.5	49.5	50.1	52.2	54.9	59.0
80.00	53.2	53.2	53.2	53.5	54.5	57.9	61.9	66.3	70.3	75.0
90.00	53.3	53.3	53.5	54.1	55.8	60.6	65.8	70.9	75.3	80.3
100.00	53.3	53.3	53.6	54.3	56.2	61.5	67.0	72.4	76.9	82.0

8 Step 7

Step 7 has a similar goal to step 6 except for the fact that now we must discriminate between customer classes. Differently from step 4, here we don't have to discover the contexts, we use the context structure found during step 4. The learning proceeds analogously to step 6 with the technique of axis projection. Since we are discriminating on the features, all the data we record is saved in a dictionary which keys are the feature combinations. The environment is still an instance of *JointBanditEnvironment*, but we will use the method *pull_arm_discriminating* to obtain disaggregated data.

In order to perform this step we defined the class *OptimalJointDiscriminatingLearner*, which upon creation accepts a *context structure*. Just like in *UCBOptimalPriceDiscriminatingLearner*, the strategy selection is deferred to the contexts and the resulting strategy is obtained by merging the results:

```
def choose_next_strategy(self):
    strategy_price = {}
    strategy_bid = {}

    for context in self.context_structure:
        arm_p, arm_b = context.choose_next_arm(self.security, self.current_round)

        for comb in context.features:
            strategy_price[comb] = arm_p
            strategy_bid[comb] = arm_b

    return strategy_price, strategy_bid
```

As for the pull of the arms and the storage of the data we have:

```
def pull_from_env(self, strategy_price, strategy_bid):
    # Actual pull
    auctions, new_clicks, purchases, tot_cost, \
        (past_strategies, visits) = self.env.pull_arm_discriminating(strategy_price,
                                                                      strategy_bid)

    # Current round data update
    for comb in self.env.get_features_combinations():
        self.tot_auctions_per_bid[comb][strategy_bid[comb]] += auctions[comb]

        self.new_clicks[comb][strategy_price[comb]][strategy_bid[comb]]\
            .append(new_clicks[comb])
        self.purchases[comb][strategy_price[comb]][strategy_bid[comb]]\
            .append(purchases[comb])
        self.tot_cost_per_bid[comb][strategy_bid[comb]] += tot_cost[comb]
```

```python
    # Past round data update
    if past_strategies[0]:
        price_strat, bid_strat = past_strategies

        for comb in self.env.get_features_combinations():
            old_arm_p = price_strat[comb]
            old_arm_b = bid_strat[comb]
            self.future_visits[comb][old_arm_p][old_arm_b].append(visits[comb])

    # Update context data
    for context in self.context_structure:
        context.merge_all_data(self.future_visits, self.purchases, self.new_clicks,
                               self.tot_cost_per_bid,self.tot_auctions_per_bid)
        context.update_pulled_arms(strategy_price, strategy_bid)

    # Update history
    self.current_round += 1
    self.strategies.append((strategy_price, strategy_bid))
```

As we can see from the code, we have to store the data accordingly to the dictionary keys.

8.1   The class *JointContext*

This class is the core of step 7 as it contains the logic to choose the most promising arm to pull for the specific context. The basic mechanism relies in the merge of the data. Basically we feed all the aggregated data and the context will extract only what is relevant for him (based on the feature combination). Due to the compexity of the data (indexed by both price and bid, with the addition of the feature combination) we had to re-implement the merge method for the double indexed case.

```python
def merge_all_data(self, future_visits_per_comb, purchases_per_comb,
                   new_clicks_per_comb, tot_cost_per_comb, tot_auctions_per_comb):

    self.future_visits = self.merge_double_indexed(future_visits_per_comb)
    self.purchases = self.merge_double_indexed(purchases_per_comb)
    self.new_clicks = self.merge_double_indexed(new_clicks_per_comb)
    self.tot_cost_per_bid = self.merge_single_indexed_bid(tot_cost_per_comb)
    self.tot_auctions_per_bid = self.merge_single_indexed_bid(tot_auctions_per_comb)

def merge_double_indexed(self, data_per_comb):
    data = []
    # merged_data[price_arm][price_bid][realization_n]
    merged_data = [[[]
```

```python
                for b in range(self.n_arms_bid)]
            for p in range(self.n_arms_price)]

    for comb in self.features:
        row = data_per_comb[comb]
        data.append(row)

    # cycle through all prices and bids
    for p in range(self.n_arms_price):
        for b in range(self.n_arms_bid):
            for i in range(len(data[0][p][b])):
                # j-th combination, i-th realization
                merged_data[p][b].append(np.sum([data[j][p][b][i]
                                                 for j in range(len(data))]))

    return merged_data

def merge_single_indexed_bid(self, data_per_comb):
    merged_data = np.sum([data_per_comb[comb] for comb in self.features], axis=0)
    return merged_data
```

Given this merge, all the following methods are very similar to step 6, since now we are working with fake aggregated data, it's like the context sees the merged data as aggregated and can thus work normally. Indeed the routine to choose the arm is:

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

def compute_safe_arms(self, arm_price, security):
    new_c = [[] for i 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])

    means = np.array([np.mean(new_c[b]) for b in range(self.n_arms_bid)])
```

```python
    std_devs = np.array([np.std(new_c[b]) for b in range(self.n_arms_bid)])
    std_devs = np.where(std_devs > 0, std_devs, 1)

    lower_security_value = np.array([norm.ppf(security, m, std)
                                     for m, std in zip(means, std_devs)])
    lower_security_value = np.where(lower_security_value > 0, lower_security_value, 0)

    expected_profits = self.compute_expected_profits_fixed_price(arm_price,
                                                                 lower_security_value)
    arm_mask = expected_profits > 0

    return arm_mask
```

We can still see the safe arm computation with some fixes to avoid nasty zero-divisions in case a bid for a specific context yield a value of 0 for *new_clicks* and consequently *tot_cost*.

8.2   The class *UCBJointContext*

This class implements the abstract methods of *JointContext*, namely *compute_projection_conversion_rates* and *compute_projection_auction_winning_probability*, just like in the previous steps. Remember that the context thanks to the merging can work on fake aggregated data, thus this two methods are very similar to the previous ones, with just some additions to avoid nasty corner cases:

```python
def compute_projection_conversion_rates(self, current_round):
    averages = self.compute_conversion_rates_averages()
    radia = self.compute_conversion_rates_radii(current_round)
    return averages + radia


def compute_projection_auction_winning_probability(self, current_round):
    averages = self.compute_auction_winning_probability_averages()
    radia = self.compute_auction_winning_probability_radii(current_round)
    return averages + radia
```

The radii of these estimates are computed as follows:

```python
def compute_conversion_rates_radii(self, current_round):
    tot_clicks_per_arm = np.array([np.sum(sum_ragged_matrix(self.new_clicks[p]))
                                   for p in range(self.n_arms_price)])
    tot_clicks_per_arm = np.where(tot_clicks_per_arm > 0, tot_clicks_per_arm, 1)

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


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

8.3 Experiment 1
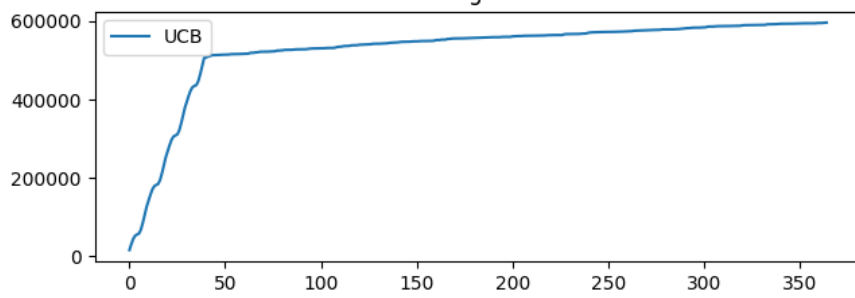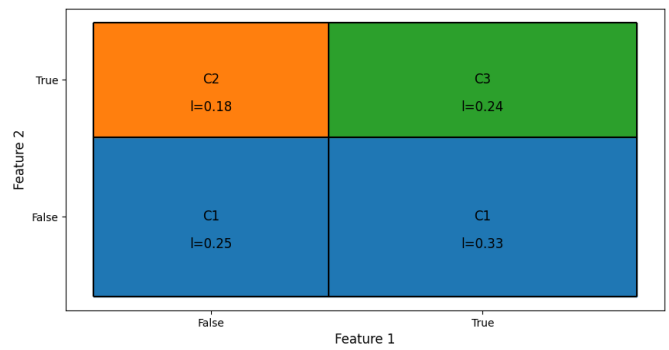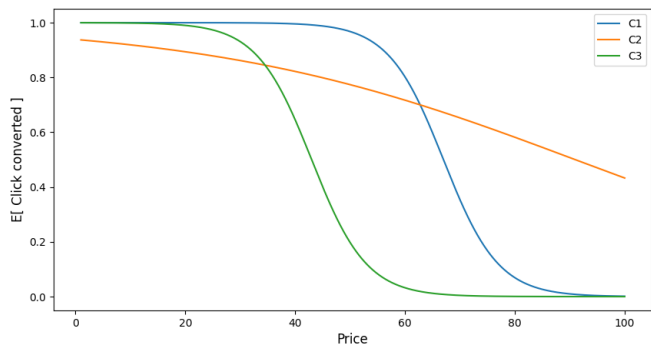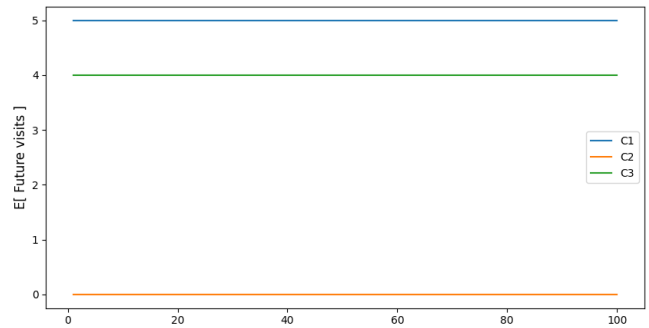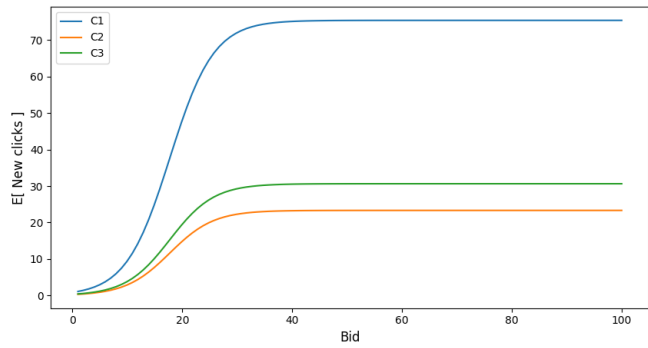
Seed: 3511939391
Context TF, FF

UCB number of pulls:
```
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|   P\B  | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|  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 |     0 |     0 |     0 |
|  50.00 |    0 |    0 |     0 |     0 |     7 |     5 |     0 |     0 |     0 |     0 |
|  60.00 |    0 |    0 |     0 |     0 |    21 |   183 |    77 |    26 |    10 |     4 |
|  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: 60.00, Optimal bid: 33.00

Gaps normalized w.r.t to maximum gap:
```
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|   P\B  | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|  10.00 | 67.7 | 68.5 |  72.6 |  80.2 |  87.6 |  91.0 |  93.6 |  95.6 |  97.8 | 100.0 |
|  20.00 | 67.4 | 67.1 |  66.2 |  65.8 |  67.1 |  68.9 |  71.2 |  73.1 |  75.3 |  77.5 |
|  30.00 | 67.1 | 65.7 |  59.9 |  51.5 |  46.6 |  46.9 |  48.8 |  50.6 |  52.8 |  55.0 |
|  40.00 | 66.7 | 64.2 |  53.6 |  37.2 |  26.3 |  25.1 |  26.6 |  28.3 |  30.5 |  32.7 |
|  50.00 | 66.4 | 62.9 |  47.9 |  24.3 |   7.9 |   5.3 |   6.3 |   8.1 |  10.2 |  12.4 |
|  60.00 | 66.3 | 62.6 |  46.3 |  20.9 |   3.0 |   0.0 |   1.0 |   2.7 |   4.8 |   7.0 |
|  70.00 | 66.9 | 64.9 |  56.6 |  44.1 |  36.1 |  35.6 |  37.2 |  39.0 |  41.2 |  43.4 |
|  80.00 | 67.4 | 67.0 |  66.0 |  65.2 |  66.2 |  68.0 |  70.2 |  72.1 |  74.3 |  76.5 |
|  90.00 | 67.5 | 67.6 |  68.3 |  70.5 |  73.8 |  76.1 |  78.5 |  80.4 |  82.6 |  84.8 |
| 100.00 | 67.5 | 67.7 |  68.7 |  71.4 |  75.1 |  77.5 |  79.9 |  81.8 |  84.0 |  86.2 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
```
Context TT

UCB number of pulls:
```
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|   P\B  | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
|  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 |     1 |     2 |     0 |     0 |     0 |     0 |     0 |
|  40.00 |    0 |    0 |     0 |     5 |   257 |    52 |     5 |     0 |     0 |     0 |
|  50.00 |    0 |    0 |     0 |     4 |     7 |     0 |     0 |     0 |     0 |     0 |
|  60.00 |    0 |    0 |     0 |     0 |     0 |     4 |     0 |     0 |     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: 40.00, Optimal bid: 27.00

Gaps normalized w.r.t to maximum gap:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	36.6	38.0	45.3	59.4	73.7	80.5	86.1	90.4	95.2	100.0
20.00	36.1	35.6	34.8	35.5	39.5	43.8	48.7	52.9	57.7	62.5
30.00	35.5	33.4	25.1	13.7	8.4	10.4	14.7	18.7	23.5	28.3
40.00	35.4	32.8	22.5	7.8	0.0	1.3	5.5	9.5	14.3	19.1
50.00	35.9	35.0	31.8	28.9	30.1	33.6	38.4	42.5	47.3	52.1
60.00	36.2	36.2	37.4	41.6	48.2	53.2	58.3	62.4	67.3	72.1
70.00	36.3	36.5	38.7	44.4	52.3	57.5	62.7	66.9	71.7	76.5
80.00	36.3	36.6	38.9	44.9	53.0	58.3	63.4	67.7	72.5	77.3
90.00	36.3	36.6	39.0	45.0	53.1	58.4	63.6	67.8	72.6	77.4
100.00	36.3	36.6	39.0	45.0	53.1	58.4	63.6	67.8	72.6	77.4

Context FT

UCB number of pulls:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
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	0	0	0
50.00	0	0	0	0	4	0	0	0	0	0
60.00	0	1	4	2	0	4	0	0	0	0
70.00	0	0	0	12	28	0	4	0	0	0
80.00	0	0	0	7	64	0	0	4	0	0
90.00	0	0	2	60	12	0	0	0	4	0
100.00	0	0	2	38	93	0	0	0	0	4

Optimal price: 90.00, Optimal bid: 27.00

Gaps normalized w.r.t to maximum gap:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	21.4	22.3	28.2	41.4	57.6	67.2	76.3	83.4	91.8	100.0
20.00	21.3	21.5	24.4	32.9	45.4	54.2	62.9	70.1	78.4	86.6
30.00	21.1	20.7	21.0	24.9	34.1	41.9	50.5	57.6	65.9	74.2
40.00	20.9	20.0	17.8	17.8	23.8	30.9	39.3	46.4	54.7	62.9
50.00	20.8	19.4	15.1	11.6	15.1	21.5	29.7	36.7	45.0	53.2
60.00	20.7	18.9	12.9	6.7	8.1	14.0	22.0	29.0	37.3	45.5
70.00	20.5	18.5	11.3	3.3	3.1	8.7	16.7	23.6	31.9	40.1
80.00	20.5	18.3	10.5	1.4	0.4	5.8	13.7	20.8	29.0	37.2
90.00	20.5	18.3	10.4	1.1	0.0	5.4	13.3	20.3	28.5	36.7
100.00	20.5	18.5	10.9	2.3	1.6	7.1	15.0	22.0	30.2	38.5

8.4   Experiment 2
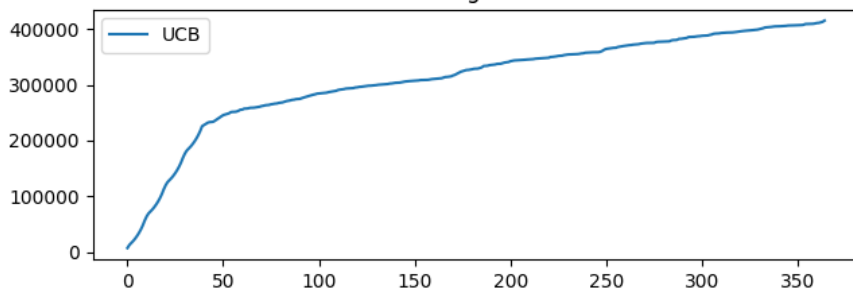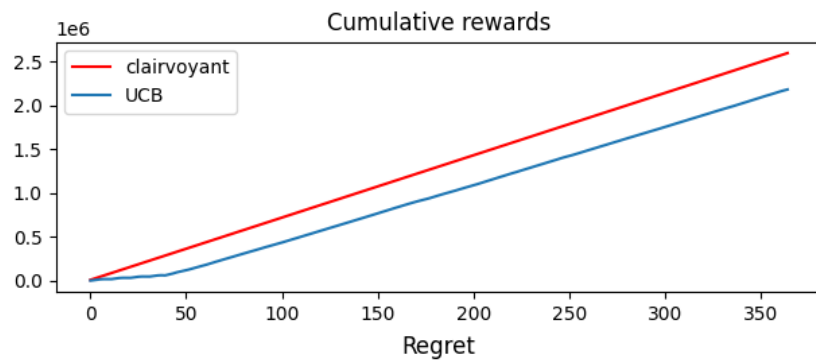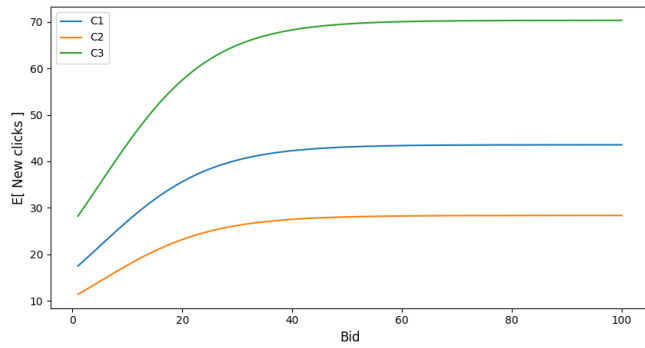
Seed: 1740212098
Context TF

UCB number of pulls:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
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	78	167	54	4	0	0	0	0
40.00	0	14	0	4	0	0	0	0	0	0
50.00	3	1	0	0	4	0	0	0	0	0
60.00	6	0	0	0	0	4	0	0	0	0
70.00	0	0	0	0	0	0	4	0	0	0
80.00	2	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: 30.00, Optimal bid: 20.00

Gaps normalized w.r.t to maximum gap:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	45.0	48.8	55.0	61.2	68.6	74.7	81.6	87.3	93.7	100.0
20.00	24.6	20.9	18.9	19.6	22.8	26.8	32.3	37.3	43.3	49.4
30.00	15.0	7.7	1.8	0.0	1.2	4.2	9.0	13.7	19.5	25.4
40.00	23.8	19.8	17.4	18.0	21.0	24.9	30.3	35.3	41.3	47.3
50.00	35.4	35.7	38.1	41.7	47.1	52.3	58.5	63.8	70.1	76.2
60.00	40.6	42.8	47.2	52.2	58.7	64.4	70.9	76.5	82.8	89.0
70.00	42.2	45.0	50.0	55.5	62.3	68.1	74.8	80.4	86.8	93.0
80.00	42.6	45.6	50.8	56.4	63.3	69.2	75.9	81.5	87.9	94.1
90.00	42.8	45.7	51.0	56.6	63.5	69.5	76.2	81.8	88.1	94.4
100.00	42.8	45.8	51.1	56.7	63.6	69.5	76.2	81.8	88.2	94.5

Context FF

UCB number of pulls:

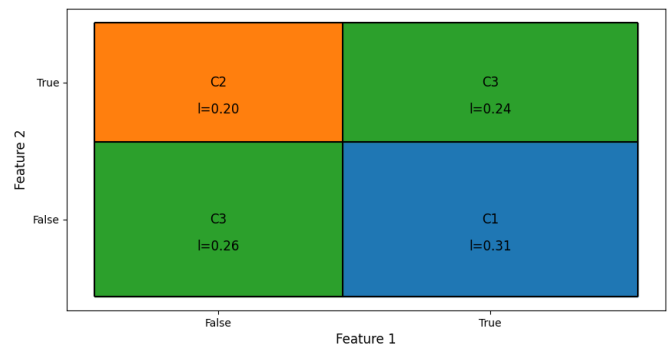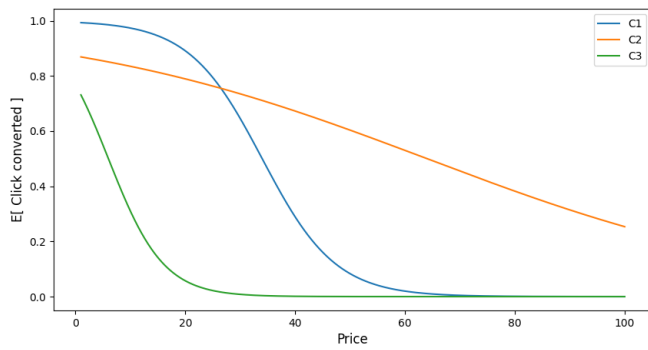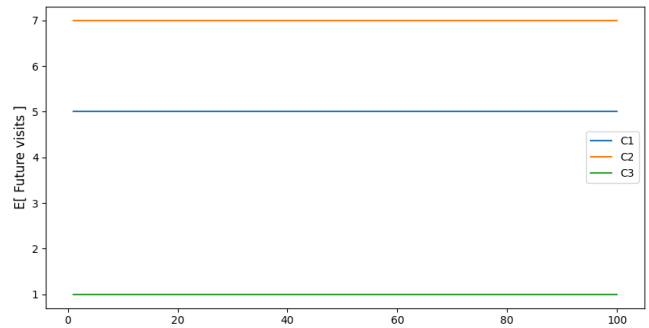| P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	4	0	0	0	0	0	0	0	0	0
20.00	6	4	0	0	0	0	0	0	0	0
30.00	0	0	4	0	0	0	0	0	0	0
40.00	4	0	0	4	0	0	0	0	0	0
50.00	11	0	0	0	4	0	0	0	0	0
60.00	26	0	0	0	0	4	0	0	0	0
70.00	41	0	0	0	0	0	4	0	0	0
80.00	61	0	0	0	0	0	0	4	0	0
90.00	78	0	0	0	0	0	0	0	4	0
100.00	98	0	0	0	0	0	0	0	0	4

Optimal price: 20.00, Optimal bid: 1.00

Gaps normalized w.r.t to maximum gap:
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
| P\B | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
10.00	1.5	7.3	17.6	28.3	41.5	52.8	65.5	76.1	88.2	100.0
20.00	0.0	5.2	14.9	25.2	38.1	49.2	61.8	72.4	84.4	96.2
30.00	0.6	6.2	16.1	26.5	39.6	50.8	63.4	74.0	86.0	97.9
40.00	0.8	6.5	16.5	27.0	40.1	51.3	64.0	74.6	86.6	98.5
50.00	0.9	6.6	16.6	27.1	40.2	51.4	64.1	74.7	86.7	98.5
60.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
70.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.8	98.6
80.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.8	98.6
90.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.8	98.6
100.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.8	98.6
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+

Context TT

UCB number of pulls:
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
| P\B | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
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	26	0	4	0	0	0	0	0	0	0
40.00	2	0	0	4	0	0	0	0	0	0
50.00	12	0	0	0	4	0	0	0	0	0
60.00	24	0	0	0	0	4	0	0	0	0
70.00	35	0	0	0	0	0	4	0	0	0
80.00	52	0	0	0	0	0	0	4	0	0
90.00	78	0	0	0	0	0	0	0	4	0
100.00	96	0	0	0	0	0	0	0	0	4
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+

Optimal price: 20.00, Optimal bid: 1.00

Gaps normalized w.r.t to maximum gap:
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
| P\B | 1.00 | 7.00 | 14.00 | 20.00 | 27.00 | 33.00 | 40.00 | 46.00 | 53.00 | 60.00 |
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+
10.00	1.5	7.4	17.7	28.3	41.5	52.8	65.5	76.1	88.2	100.0
20.00	0.0	5.3	14.9	25.2	38.1	49.2	61.8	72.3	84.4	96.2
30.00	0.6	6.2	16.1	26.6	39.6	50.8	63.4	74.0	86.0	97.9
40.00	0.8	6.5	16.5	27.0	40.1	51.3	64.0	74.5	86.6	98.4
50.00	0.9	6.6	16.6	27.1	40.2	51.4	64.1	74.7	86.7	98.6
60.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
70.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
80.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
90.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
100.00	0.9	6.6	16.6	27.2	40.2	51.4	64.1	74.7	86.7	98.6
+--------+------+------+-------+-------+-------+-------+-------+-------+-------+-------+

Context FT

UCB number of pulls:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
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	0	0	0
50.00	0	0	1	0	9	3	0	0	0	0
60.00	0	0	0	1	4	16	9	6	4	4
70.00	0	0	0	0	4	15	19	17	9	4
80.00	0	0	1	6	10	17	20	17	6	4
90.00	0	0	0	3	5	14	29	17	17	7
100.00	0	0	2	12	16	12	3	2	0	4

Optimal price: 70.00, Optimal bid: 40.00

Gaps normalized w.r.t to maximum gap:

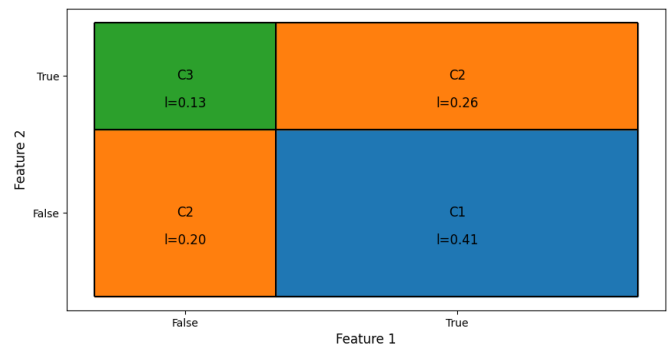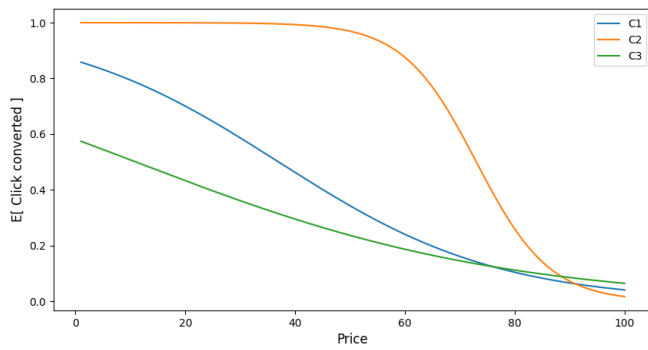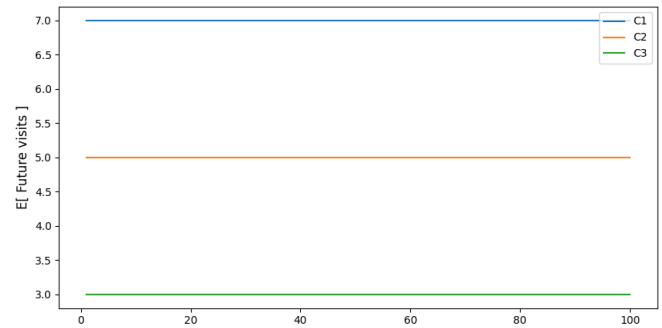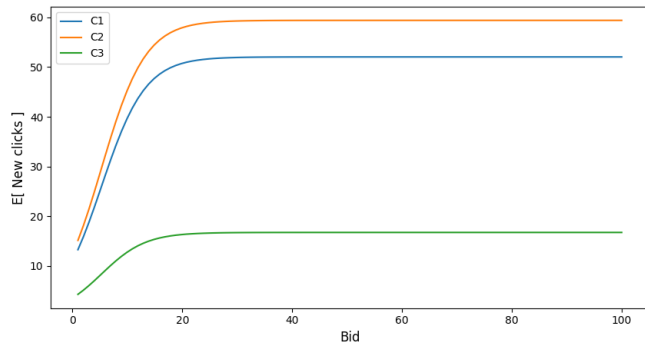| P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	80.2	81.6	84.0	86.2	88.9	91.1	93.5	95.5	97.8	100.0
20.00	69.1	66.5	64.3	63.7	64.1	65.1	66.7	68.4	70.4	72.5
30.00	59.5	53.3	47.4	44.1	42.5	42.6	43.5	44.9	46.7	48.7
40.00	51.8	42.7	33.7	28.4	25.2	24.4	24.8	25.9	27.6	29.5
50.00	46.1	35.0	23.7	16.9	12.6	11.2	11.2	12.1	13.7	15.5
60.00	42.7	30.4	17.7	10.0	4.9	3.2	3.0	3.8	5.3	7.1
70.00	41.5	28.7	15.5	7.5	2.2	0.3	0.0	0.8	2.2	4.0
80.00	42.2	29.6	16.7	8.8	3.7	1.8	1.6	2.4	3.9	5.7
90.00	44.3	32.5	20.5	13.2	8.5	6.9	6.8	7.7	9.2	11.0
100.00	47.5	36.9	26.1	19.7	15.6	14.4	14.5	15.4	17.0	18.9

8.5 Experiment 3
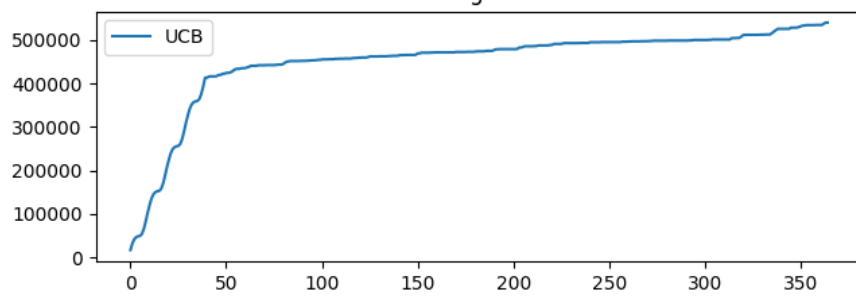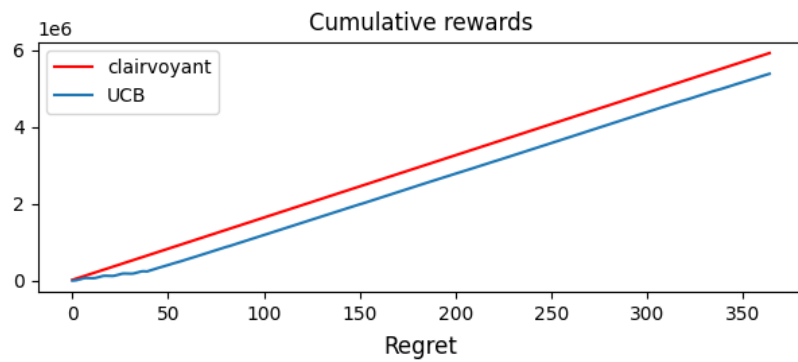
Seed: 4059059292
Context TT, TF, FT, FF

UCB number of pulls:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
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	0	0	0
50.00	0	0	5	14	4	0	0	0	0	0
60.00	0	0	14	215	47	14	2	0	0	0
70.00	0	0	11	7	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: 60.00, Optimal bid: 20.00

Gaps normalized w.r.t to maximum gap:

P\B	1.00	7.00	14.00	20.00	27.00	33.00	40.00	46.00	53.00	60.00
10.00	65.1	72.3	80.3	84.2	87.3	89.7	92.4	94.7	97.3	100.0
20.00	58.4	56.4	56.6	58.4	61.0	63.3	66.0	68.3	71.0	73.6
30.00	52.7	42.7	36.3	36.4	38.5	40.7	43.4	45.7	48.3	51.0
40.00	48.0	31.7	19.9	18.6	20.4	22.6	25.2	27.5	30.1	32.8
50.00	44.7	23.7	8.0	5.7	7.2	9.3	11.9	14.2	16.8	19.5
60.00	43.2	20.2	2.8	0.0	1.4	3.5	6.1	8.4	11.0	13.7
70.00	45.8	26.5	12.2	10.2	11.8	13.9	16.5	18.8	21.5	24.1
80.00	52.3	41.7	34.8	34.8	36.9	39.1	41.8	44.0	46.7	49.4
90.00	56.9	52.6	51.0	52.4	54.9	57.1	59.8	62.1	64.7	67.4
100.00	58.8	57.3	57.9	59.9	62.5	64.8	67.5	69.8	72.4	75.1

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

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

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

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

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

therefore

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

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

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

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

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

$$
= \mathbb{E}_{n_{c,b}}\left[\sum_{c \in C} \sum_{i=1}^{n_{c,b}} \left(\mathbb{E}\left[D_{c,p,i}(1 + F_{c,i})m(p) \middle| N_{c,b} = n_{c,b} \right] - k(c,b) \right) \right]
$$

$$
= \mathbb{E}_{n_{c,b}}\left[\sum_{c \in C} \sum_{i=1}^{n_{c,b}} \left(r(c,p)(1 + f(c))m(p) - k(c,b) \right) \right]
$$

$$
= \mathbb{E}_{n_{c,b}}\left[\sum_{c \in C} n_{c,b} \left(r(c,p)(1 + f(c))m(p) - k(c,b) \right) \right]
$$

$$
= \sum_{c \in C} \mathbb{E}_{n_{c,b}}\left[n_{c,b} \left(r(c,p)(1 + f(c))m(p) - k(c,b) \right) \right]
$$

$$
= \sum_{c \in C} \mathbb{E}_{n_{c,b}}\left[n_{c,b} \right] \left(r(c,p)(1 + f(c))m(p) - k(c,b) \right)
$$

$$
= \sum_{c \in C} \mathbb{E}\left[N_{c,b} \right] \left(r(c,p)(1 + f(c))m(p) - k(c,b) \right)
$$

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

D Proof of the Bid Independent Price Hierarchy lemma

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

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

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

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.