Project Presentation

Online Learning Application
Academic Year 2021-2022
Prof. GATTI NICOLA

Aygalic Jara
Christopher Volpi
Giulia Montani
Luca Mainini
Roberta Troilo

Scenario

E commerce

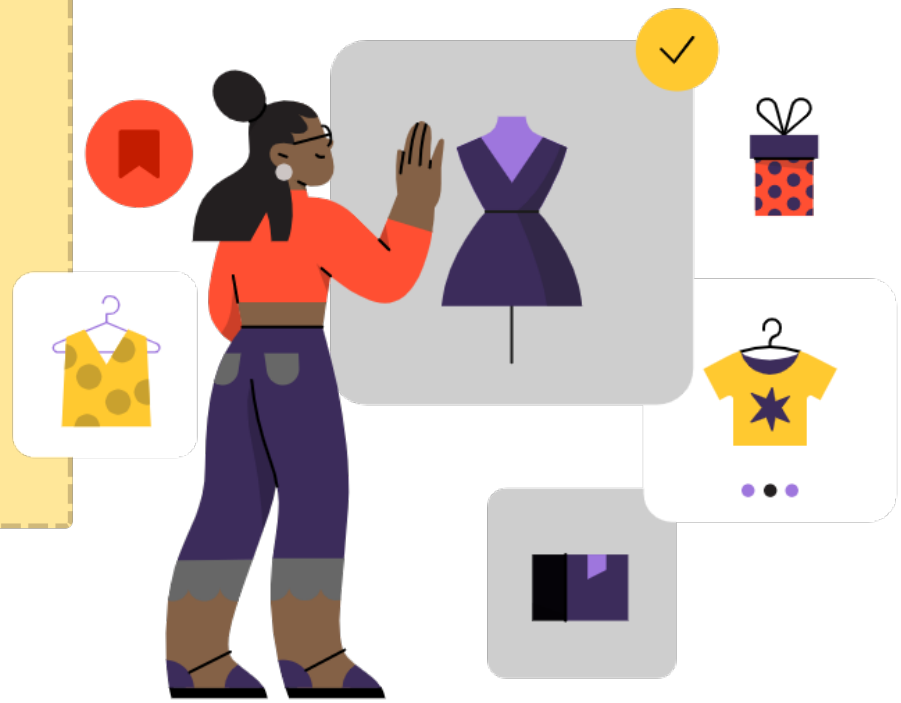
Italian Based cloth retailer

Italian

Non Italian

Member

Regular



1 – Environment

Architecture

- User
- Product
- Daily Customer
- Ecommerce

User

Base Class

reservation_price
product_clicked
cart
quantities
P (influence graph)
primary

User0

Non italian

- Higher reservation price
- More interested in shoes, shirt and pants
- Willing to pay more for more expensive products (shipping costs are to be added)

User1

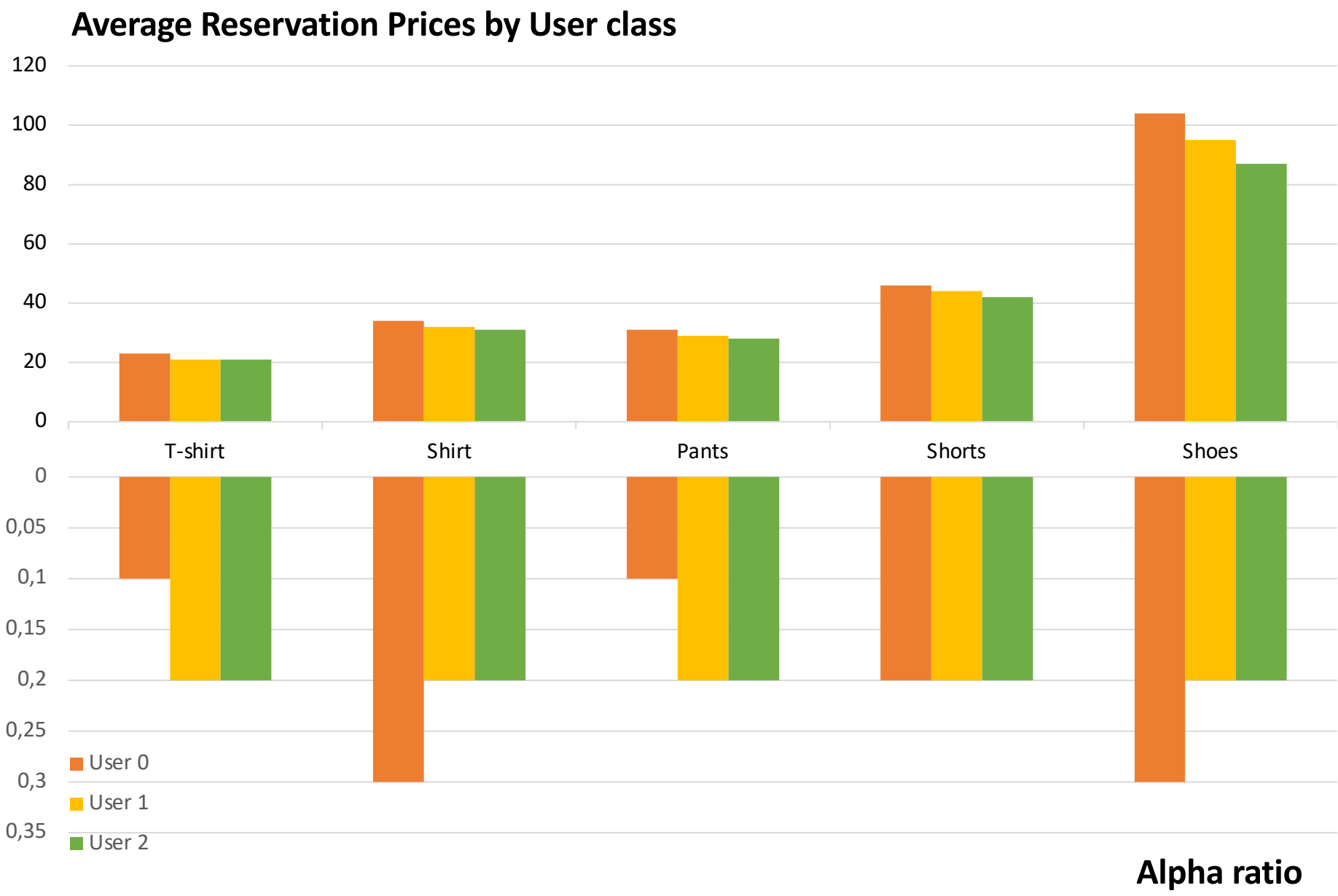
Non Member italian Customer

- Average reservation price
- Equally interested in all products

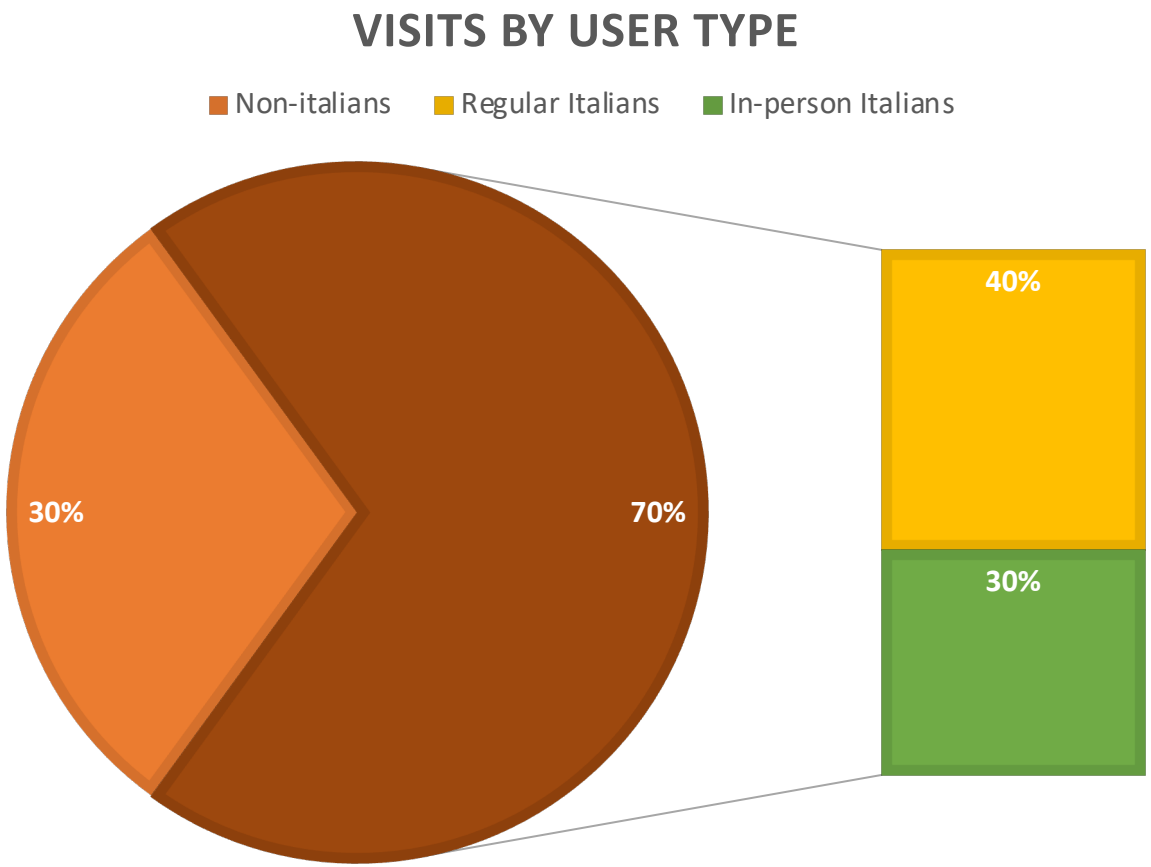
User2

Italian Member

- Lower reservation price
- Used to membership discounts
- They buy but with lower prices


Average Reservation Prices by User class
120
100
80
60
40
20
0
T-shirt
Shirt
Pants
Shorts
Shoes
0
0,05
0,1
0,15
0,2
0,25
0,3
0,35
User 0
User 1
User 2
Alpha ratio

User Distribution*



*based on previous year's data

Product, Pricing & Suggestion

T-shirt

Price	*Margin*
19.99	**20%**
21.99	
23.99	
26.99	

Secondary Products
Shorts
Pants

Shorts

Price	
25.99	**30%**
28.99	
31.99	
34.99	

Secondary Products
T-shirt
Shirt

Shirt

Price	
29.99	**20%**
32.59	
35.99	
39.99	

Secondary Products
Pants
Shorts

Pants

Price	
38.99	**40%**
43.99	
46.99	
49.99	

Secondary Products
Shirt
T-shirt

Shoes

Price	
79.99	**50%**
89.99	
99.99	
109.99	

Secondary Products
Pants
Shirt

Suggestion Interface

Daily Customers

Attributes

Users
users_distribution = [0.3, 0.4, 0.3]

users accessing the website during the day
distribution of the users among the classes

Methods

UsersGenerator

Generate daily users choosing which product they see first (if they arrive at the website) based on their type.

Hypothesis on User Distribution

The number of daily users follows a Gaussian distribution

In case we consider a homogeneous:

$$\mathrm{N} \sim \mathcal{N}(\mu_{users}, (0{,}2 * \mu_{users})^2)$$

In case we consider the 3 classes of users:

$\mathrm{N}_0 \sim \mathcal{N}(\mu_0, (0{,}2 * \mu_0)^2)$ where $\mu_0 = 0.3 * \mu_{users}$

$\mathrm{N}_1 \sim \mathcal{N}(\mu_1, (0{,}2 * \mu_1)^2)$ where $\mu_1 = 0.4 * \mu_{users}$

$\mathrm{N}_2 \sim \mathcal{N}(\mu_2, (0{,}2 * \mu_2)^2)$ where $\mu_2 = 0.3 * \mu_{users}$

$\boldsymbol{\alpha_i} = [\alpha_{1i}, \alpha_{2i}, \alpha_{3i}, \alpha_{i4}, \alpha_{5i}, \alpha_{6i}] \sim Dir(\widehat{\boldsymbol{\alpha_i}})$ where $i = 1{,}2{,}3$

users_per_product_i = np.random.multinomial($\mathrm{N_i}$, $\boldsymbol{\alpha_i}$)

Daily Customers - UsersGenerator

- def UsersGenerator(self, number_users, fixed_alpha, fixed_weights, binary_features):
  """Generate daily users choosing which product they see first (if they arrive at the website) based on their type.

  :param number_users: average number of potential users in a day
  :type number_users: int
  :param fixed_alpha: 1 if alpha is fixed (uniformly distributed over the products)
  :type fixed_alpha: bool
  :param fixed_weights: 1 if graph weights are fixed
  :type fixed_weights: bool
  :param binary_features: 1 if we distinguish between user's types, 0 if not
  :type binary_features: bool

E-commerce

Attributes

- products
- daily_users
- time_history
- lambda_
- graph
- daily_rewards
- daily_clicks
- daily_purchases
- daily_rewards_per_product
- daily_purchased_units
- binary_features

list of products
list of lists of users
history of products visited probability to check the second product
1 for the first secondary and lambda for the second
secondary total daily rewards for each day
clicks per product for the current day
purchases per product for the current day
daily rewards per product
daily purchased units per product
1 if we distinguish between users

Methods

__init__

Generate daily users choosing which product they see first (if they arrive at the website) based on their type.

simulate_day

simulate a day of visits in the website

visit

Simulates a visit on the website.

E-commerce

```
def __init__(self, binary_features=1):
    """ E-commerce class.
    :param binary_features: 1 if we distinguish between users
    """
    # list of products
    self.products = []
    # list of lists of users
    self.daily_users = []
    # dataset with the history of products visited by each user in each day
    self.time_history = []
    # probability that the user checks the second product
    self.lambda_ = 0.5
    # weight of the graph: 1 for the first secondary and lambda for the second secondary
    self.graph = np.array(
        [[0., self.lambda_, 0., 1., 0.],
         [1., 0., 0., self.lambda_, 0.],
         [0., 1., 0., self.lambda_, 0.],
         [0., self.lambda_, 1., 0., 0.],
         [1., 0., 0., self.lambda_, 0.]])
    # list of the total daily rewards for each day
    self.daily_rewards = []
    # clicks per product for the current day
    self.daily_clicks = []
    # purchases per product for the current day
    self.daily_purchases = []
    self.daily_rewards_per_product = []
    # daily purchased units per product
    self.daily_purchased_units = []
    # binary_features: 1 if we do not distinguish between users
    self.binary_features = binary_features
```

E-commerce: simulation of a day

```
def simulate_day(self, number_users, fixed_alpha, fixed_weights, fixed_units):
    """This function simulate a day of visits in the website

    :param number_users: average number of potential users in a day
    :type number_users: int
    :param fixed_alpha: 1 if alpha is fixed (uniformly distributed over the products)
    :type fixed_alpha: bool
    :param fixed_weights: 1 if graph weights are fixed
    :type fixed_weights: bool
    :param fixed_units: if 1 always the same number of units is bought for each product
    :type fixed_units: bool
    :param alpha: vector with dirichlet parameters
    :return: a Daily_Customers class
```

E-commerce: when a user visits the website

```
def visit(self, user):
```

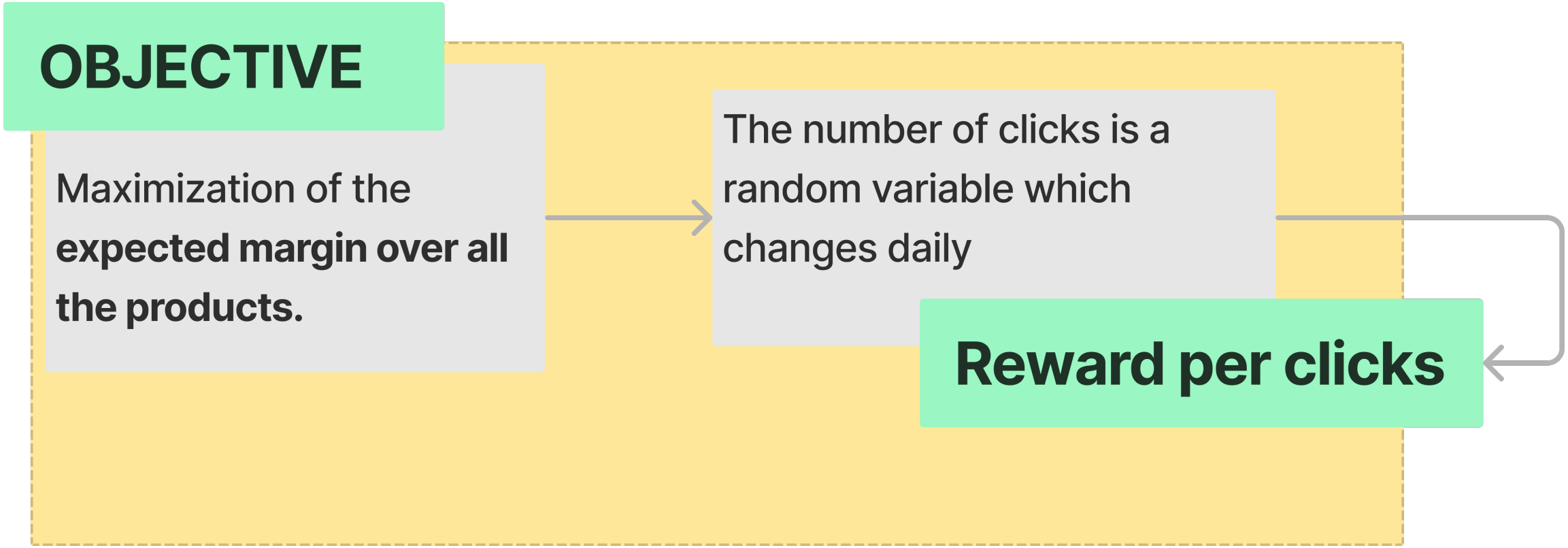
We create a graph to compute the probabilities of a click based on the slot of a product and on the user's will to buy.

```
prob_matrix = user.P * self.graph
products_clicked = p > np.random.rand(p.shape[0], p.shape[1])
```

We match the indexes, checking the user's reservation price.

```
stop_idxs = np.where(np.array(user.reservation_price)[secondary_slots] <
    prod_prices[secondary_slots])
go_idxs = np.where(np.array(user.reservation_price)[secondary_slots] >=
    prod_prices[secondary_slots])
```

2- Optimization problem & Greedy Algorithm



Optimization pricing problem

Given that:

i = index of the product (from 1 to 5)

j_i = price index for product i

X_{ij_i} = r.v. indicating the reward per click given by the i-th product pulling the j-th price

The optimization problem is

$$\max_{(j_1,j_2,j_3,j_4,j_5)} \mathbb{E}\left[\sum_{i=1}^{5} X_{ij_i}\right] = \mathbb{E}\left[\max_{j_i} \sum_{i=1}^{5} X_{ij_i}\right] = \mathbb{E}\left[\max_{j_i} \sum_{i=1}^{5} m_{ij_i} c_{ij_i} n_i\right]$$

where

m_{ij_i} = margin over product i at price j

c_{ij_i} = conversion rate for product i at price j .

n_i = number of items sold per product i

Optimization pricing problem

The assumptions of the profit maximization problem are:

- the rewards are stationary, so the parameters of the probability distributions generating the rewards are **stationary** (the market does not change in time)
- for each product
    - a single price is pulled at every time (each day)
    - given a price, we can immediately observe the samples
- the number of time points (days) is bounded (so not infinite)
- according to the specific algorithm applied, prices are updated at each round in order to minimize the cumulative regret

Greedy Algorithm

Initialize variables

Preliminary step

Loop over different prices

update the margin and the conversion rate of the kth element:
if in the previous iteration the maximum was updated it means that the kth product has increased its price by one increment (since we got a better configuration for the reward)
So we don't reset the matrix for it but we set a new increased price

Loop over different products

we compute the total expected reward as the sum of the rewards of each product, except the one which has increased his price

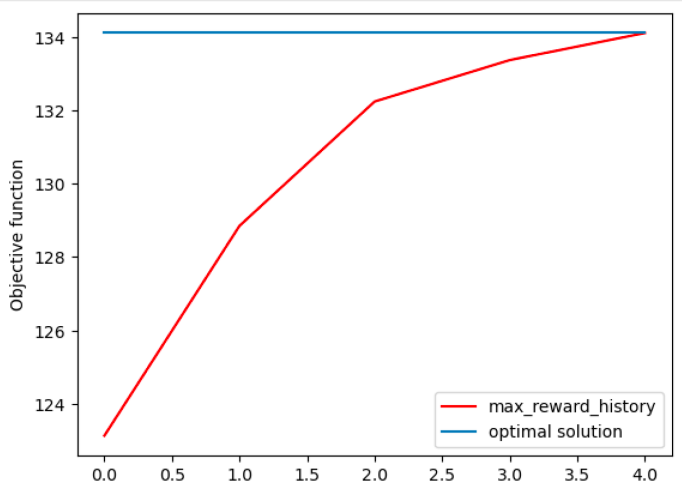
Returns the best configuration associated with the rewards collected and the maximum collected (+ number of iteration)

Greedy Algorithm

124,26	123,87	123,13	126,52	**128,85**

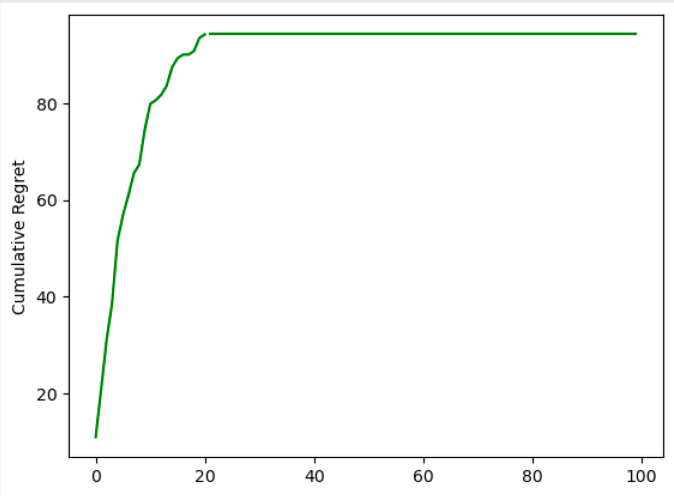
Greedy Algorithm

Performance



maximization of the cumulative expected margin over all the products

in this case the greedy algorithm reached the optimal solution but it's not always the case

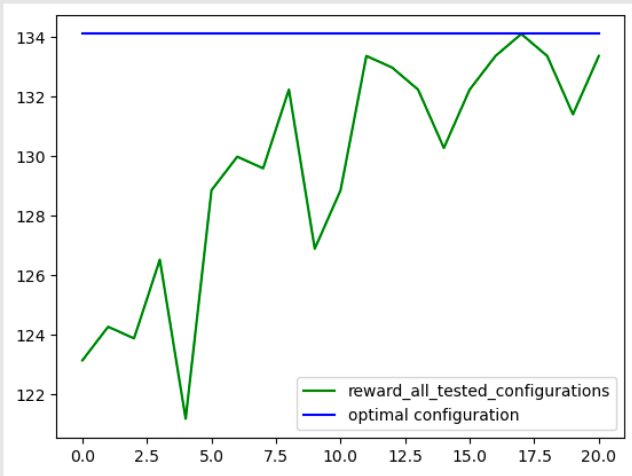


The cumulative regret plateau as we find the optimal solution

Algorithm:

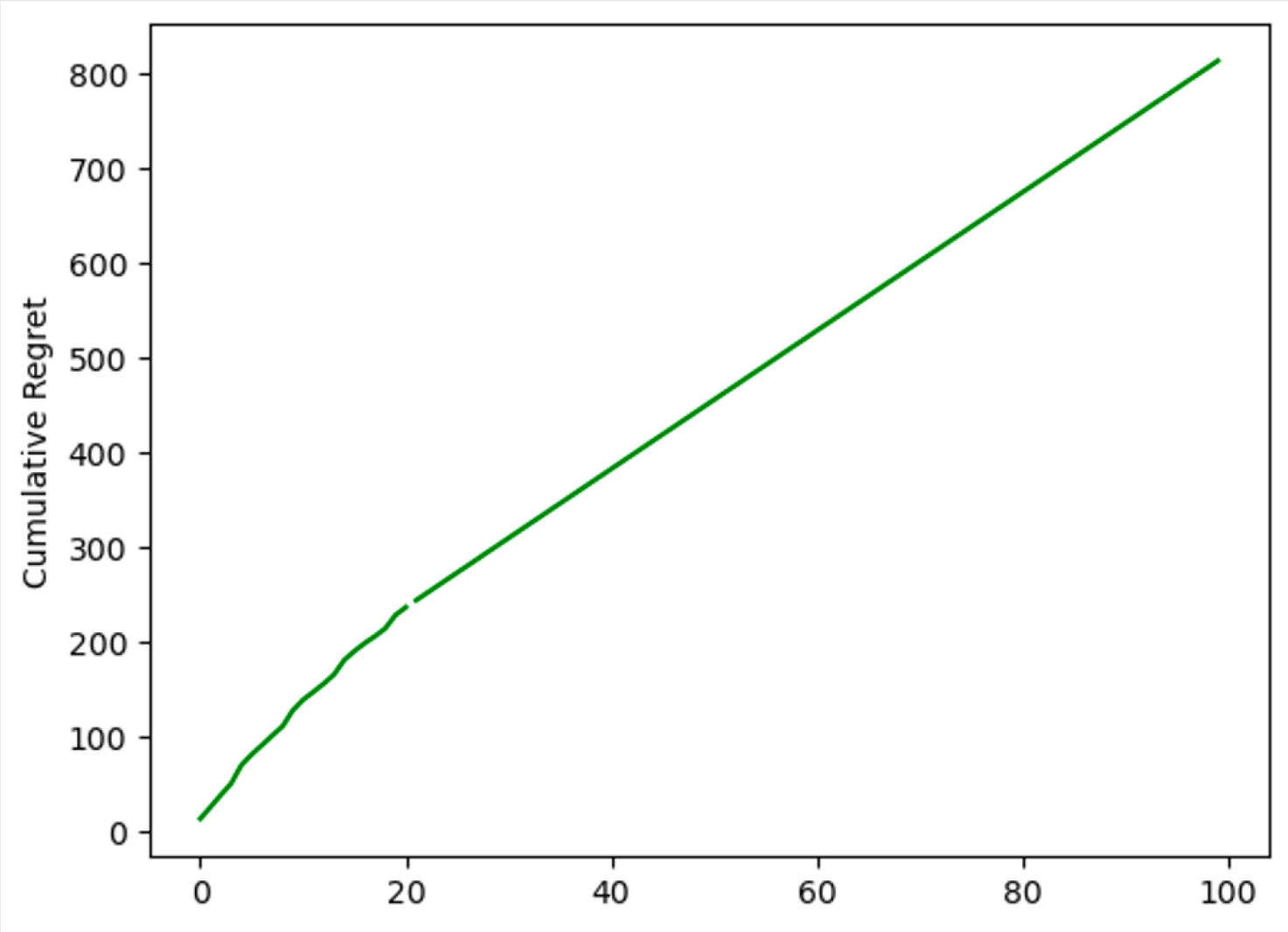
1): increase price of the product 5
2): increase price of the product 3
3): increase price of the product 1
4): increase price of the product 2

Best configuration: [1. 1. 1. 0. 1.]



starting configuration 123,13
Iteration 4 (maximum): 134,11

Non optimal scenario



Conversion is not guaranteed

Results Step 2

The greedy algorithm works better in the case the optimal prices are between the first possible arms. This is because the algorithm **does not cycle.**

Its performances are highly related to the **graph weights** and **reservation prices**.

3 – Optimization with uncertain conversion rates

Optimization with **uncertain α**

Binary features cannot be observed and therefore **data are aggregated**

Number of items sold is **known:**
[2, 1, 3, 3, 1]

Hypothesis on User Distribution

The number of daily users follows a Gaussian distribution

In case we consider a homogeneous:

$$\mathrm{N} \sim \mathcal{N}(\mu_{users}, (0{,}2 * \mu_{users})^2)$$

In case we consider the 3 classes of users:

$\mathrm{N}_0 \sim \mathcal{N}(\mu_0, (0{,}2 * \mu_0)^2)$ where $\mu_0 = 0.3 * \mu_{users}$

$\mathrm{N}_1 \sim \mathcal{N}(\mu_1, (0{,}2 * \mu_1)^2)$ where $\mu_1 = 0.4 * \mu_{users}$

$\mathrm{N}_2 \sim \mathcal{N}(\mu_2, (0{,}2 * \mu_2)^2)$ where $\mu_2 = 0.3 * \mu_{users}$

$\boldsymbol{\alpha_i} = [\alpha_{1i}, \alpha_{2i}, \alpha_{3i}, \alpha_{i4}, \alpha_{5i}, \alpha_{6i}] \sim Dir(\widehat{\boldsymbol{\alpha_i}})$ where $i = 1{,}2{,}3$

users_per_product_i = np.random.multinomial($\mathrm{N_i}$, $\boldsymbol{\alpha_i}$)

Optimization with uncertain conversion rates

STEPS:

Estimation of the conversion rates through an algorithm which pulls cyclically all the arms for each product and simulates the visits in the website.

Computation of the optimal "clairvoyant" solution per round and of the best configuration:

*Best_arm_per_product = argmax(margin_matrix*conv_rates)*

Design of the optimization algorithms: TS and UCB

Every day we compute the reward for each product in the Learning Environment:

reward[i] = E.daily_rewards_per_product[i] / clicks_current_day[i]

Learning Algorithms

Base class Learner

Methods

__init__(self, n_arms)
update_observations(self, pulled_arm, reward)
reset(self)

Specialized learner inherit from base class

Thompson sampling

upper confidence bound (UCB1)

UCB Algorithm

Methods

__init__
pull_arm
update

TS Algorithm

Methods

__init__
pull_arm
update

__init__

```
def __init__(self, n_arms, c=2):
:param n_arms: number of arms
:param c: confidence value (in class it was 2)
"""
  super().__init__(n_arms)
  self.expected_rewards = np.zeros([5, n_arms])
  self.confidence = np.array(([[np.inf] * n_arms] * 5))
  self.make_comparable = np.zeros(5)
  self.explore = np.array([0, 1, 2, 3, 0, 1, 2, 3])
  self.c = c
```

pull_arm

Pulls the correct arm to be played at the next round

```
def pull_arm(self):

if self.t < 2 * self.n_arms:
  return np.array([self.explore[self.t]] * 5)

if self.t == 2 * self.n_arms: # 2*4=8
  for i in range(5):
    self.make_comparable[i] = np.std(self.expected_rewards[i])
idx = np.zeros(5)
upper_conf = np.zeros((5, 4))

for i in range(5):
  upper_conf[i] = self.expected_rewards[i] + self.confidence[i] * self.make_comparable[i]
  idx[i] = np.random.choice(np.argwhere(upper_conf[i] == upper_conf[i].max()).reshape(-1))
return idx
```

Initial exploration

Compute & store the maximum expected reward for each product using upper confidence

Chose the arm with the highest explected reward for each product
Taking into account the confidence interval (shrinking as the product gets pulled)

update

```
def update(self, pulled_arm, reward):
  self.t += 1
  for i in range(5):
    self.counter_per_arm[i][int(pulled_arm[i])] += 1
    self.expected_rewards[i][int(pulled_arm[i])] = (
      self.expected_rewards[i][int(pulled_arm[i])] * (
      self.counter_per_arm[i][int(pulled_arm[i])] - 1) +
      reward[i]) / self.counter_per_arm[i][
      int(pulled_arm[i])]
    n_samples = np.size(self.rewards_per_arm[i][int(pulled_arm[i])])
    self.confidence[i][int(pulled_arm[i])] = (self.c * np.log( self.t) /
    n_samples) ** 0.5 if n_samples > 0 else np.inf
  self.update_observations(pulled_arm, reward)
```

UCB Algorithm

...

__init__

```
def __init__(self, n_arms):
  super().__init__(n_arms)
  self.beta_parameters = np.array([np.ones((n_arms, 2))] * 5)
  self.expected_rewards = np.zeros([5, n_arms])
```

pull_arm

Pulls the correct arm to be played
at the next round

First exploration of all possible arms

We pull the arm with the best expected

Reward drawn from a beta dist. w.r.t. the margins

```
def pull_arm(self, margins_matrix):

  if self.t < self.n_arms:
    return np.array([self.t] * 5)

  idx = np.zeros(5)
  for i in range(5):
    idx[i] = np.argmax(
    np.random.beta(self.beta_parameters[i][:, 0], self.beta_parameters[i][:, 1]) * margins_matrix[i, :])

  return idx
```

update

Update the beta distribution with the clicks
And purchases of the day: estimation
Of conversion rate

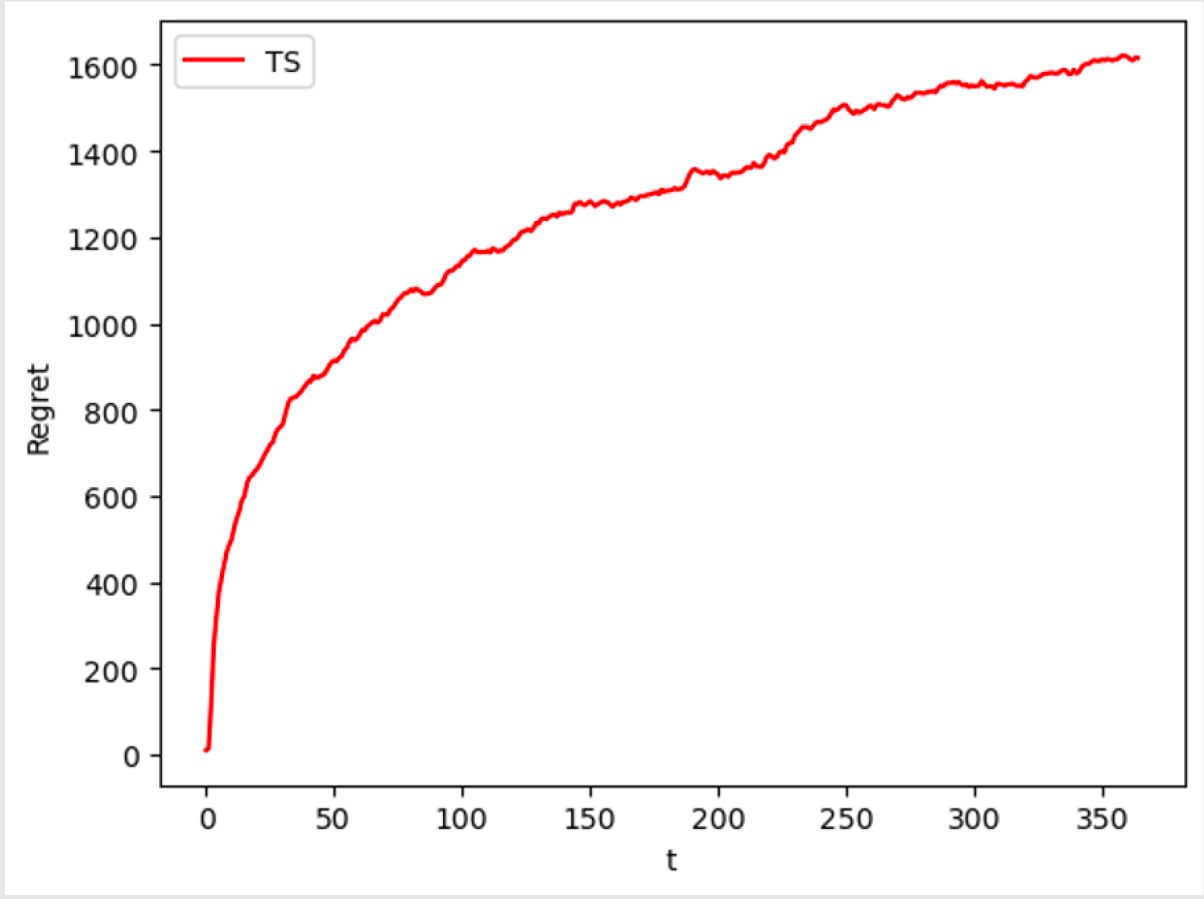
```
def update(self, pulled_arm, reward, clicks, purchases, daily_units):

  self.t += 1
  self.update_observations(pulled_arm, reward)

  for i in range(5):
    self.beta_parameters[i][int(pulled_arm[i]), 0] = self.beta_parameters[i][int(pulled_arm[i]), 0] + purchases[i]
    self.beta_parameters[i][int(pulled_arm[i]), 1] = self.beta_parameters[i][int(pulled_arm[i]), 1] + (clicks[i] - purchases[i])
    self.counter_per_arm[i][int(pulled_arm[i])] += 1
    self.expected_rewards[i][int(pulled_arm[i])] = (self.expected_rewards[i][int(pulled_arm[i])] *
                                                    (self.counter_per_arm[i][int(pulled_arm[i])] - 1) +
                                                    reward[i]) / self.counter_per_arm[i][int(pulled_arm[i])]
```

TS Algorithm

TS cumulative regret



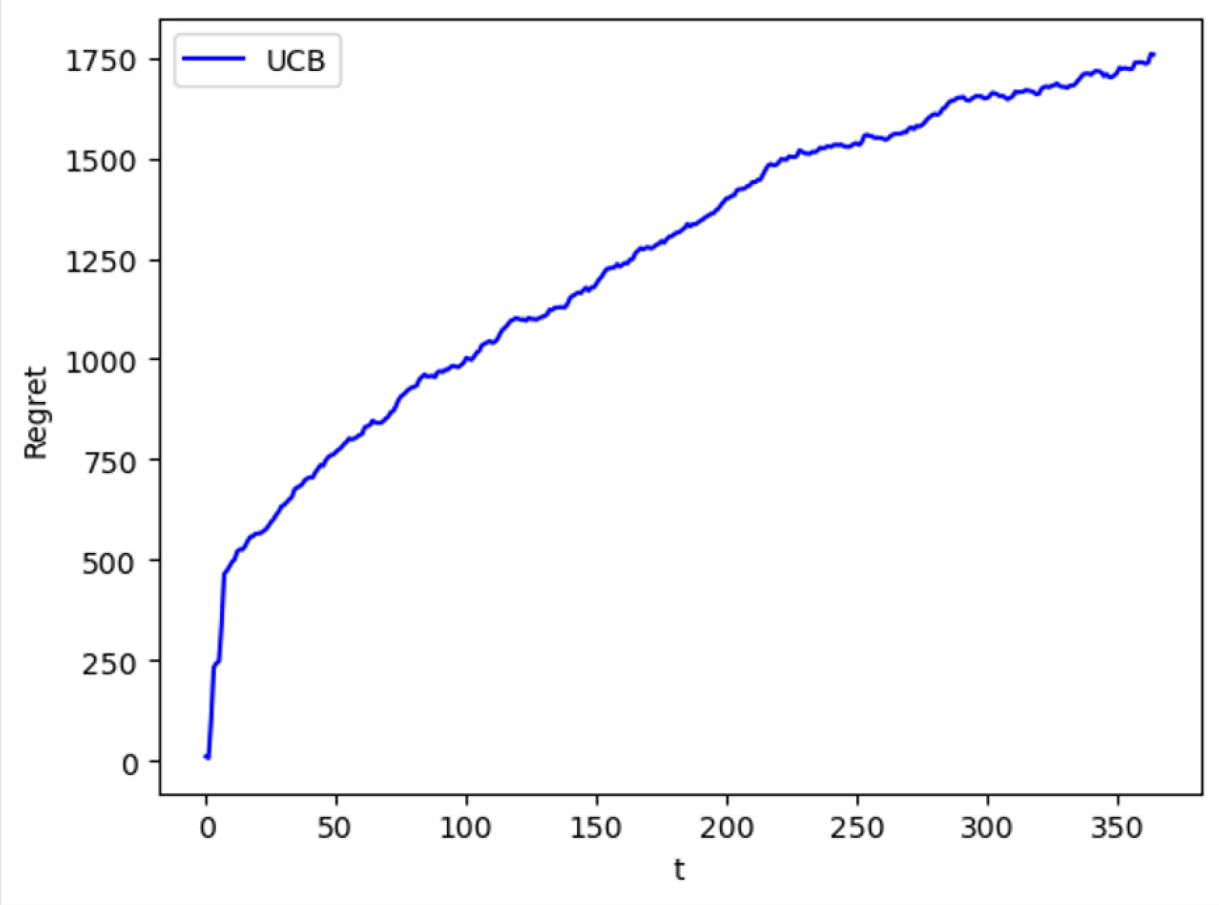
Best configuration: [1. 1. 1. 0. 1.]

One run example

Pull per arm :

$$\begin{bmatrix} 12 & 337 & 8 & 8 \\ 63 & 289 & 6 & 7 \\ 13 & 342 & 4 & 6 \\ 281 & 64 & 16 & 4 \\ 1 & 352 & 6 & 6 \end{bmatrix}$$

UCB cumulative regret



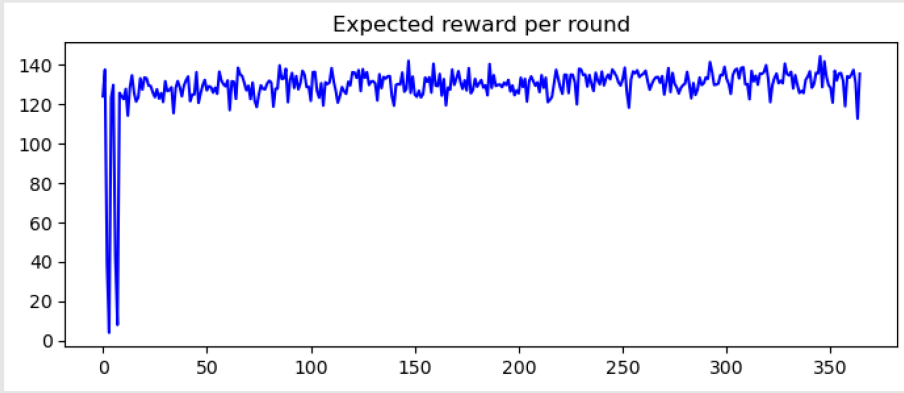
Best configuration: [1. 1. 1. 0. 1.]

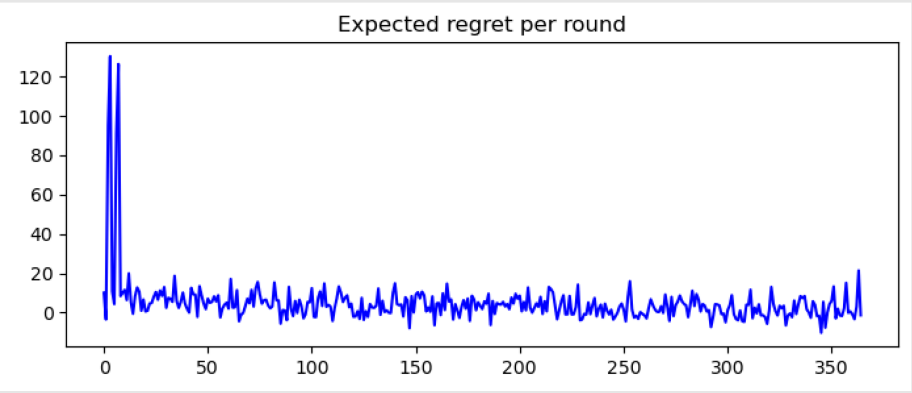
One run example

Pull per arm :

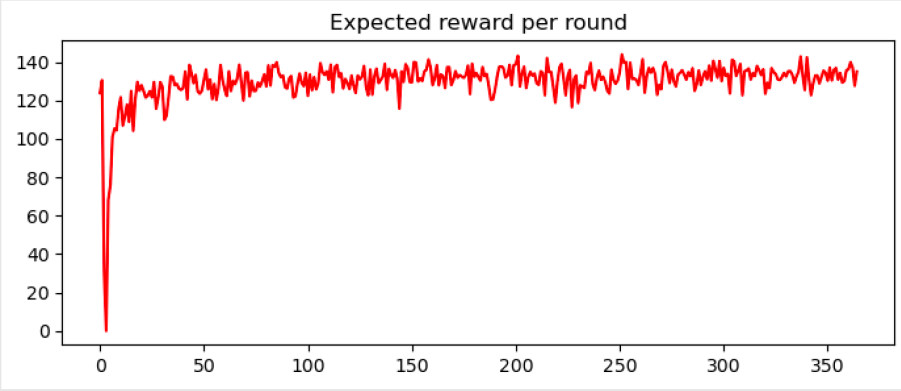
$$\begin{bmatrix} 82 & 275 & 6 & 2 \\ 82 & 264 & 17 & 2 \\ 69 & 292 & 2 & 2 \\ 217 & 139 & 7 & 2 \\ 46 & 315 & 2 & 2 \end{bmatrix}$$

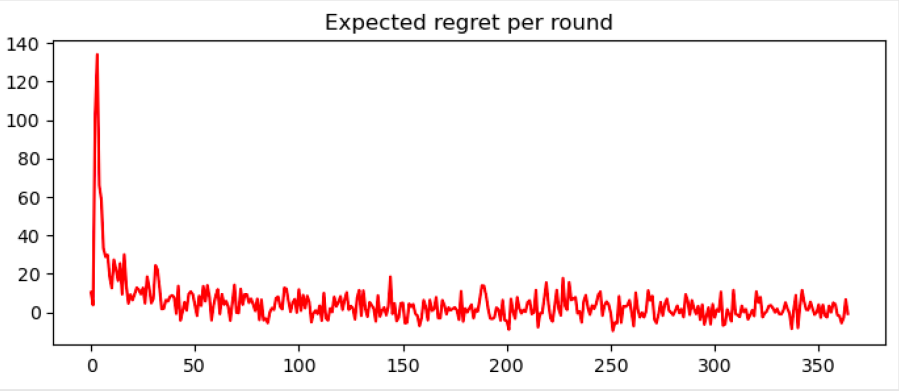
UCB Algorithm





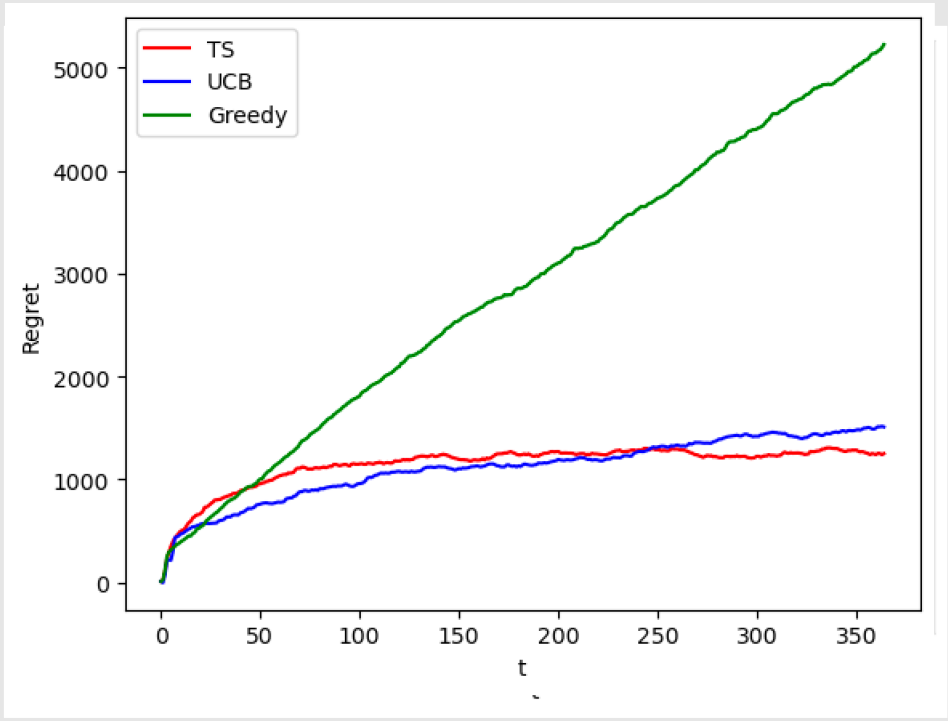
TS Algorithm





Learning Algorithms Performance

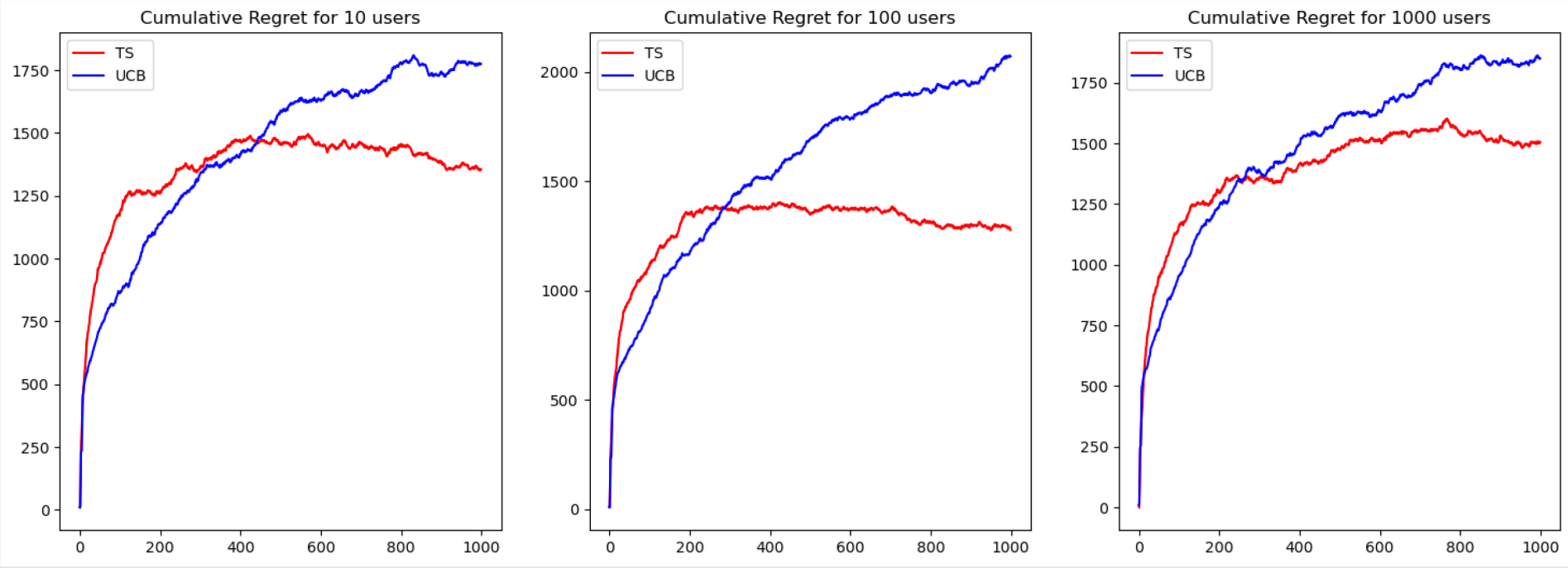
Cumulative regret over time



Learning Algorithms Performance

Cumulative regret over time depending on the number of users

10, 100, 1000 users



Theoretical bounds

UCB1 Regret

$$\Delta_a := \mu_{a^*} - \mu_a$$

$$\mathscr{R}_T(\text{UCB1}) \leq \sum_{a:\mu_a<\mu_{a^*}} \frac{4\,\log(T)}{\Delta_a} + 8\,\Delta_a$$

Thompson Sampling Regret

$$\Delta_a := \mu_{a^*} - \mu_a$$

$$\mathscr{R}_T(\text{TS}) \leq (1+\epsilon) \sum_{a:\mu_a<\mu_{a^*}} \frac{\Delta_a \left(\log(T) + \log(\log(T))\right)}{\mathscr{K}\mathscr{L}(\mu_{a^*}, \mu_a)} + C\left(\epsilon, \mu_{a_1}, \ldots, \mu_{a_{|A|}}\right)$$

Empirical regret over 365 rounds: 1758

Empirical regret over 365 rounds: 1615

Taking into account that our rewards are computed per click (margin * conv_rate * numb_units) and not drawn from a Bernoulli.

Results of step 3

Results considering a period of 365 days over 20 runs:

Expected total collected rewards with the best configuration: 48983

TS Total collected reward: **47368** with 554 of standard deviation between the experiments
UCB Total collected reward: 47225 with 401 of standard deviation between the experiments

TS cumulative expected regret: **1615**
UCB cumulative expected regret: 1758

Expected reward per round with the best configuration: 134.20

TS average expected reward per round: **129.77**
UCB average expected reward per round: 129.38

TS average expected regret per round: **4.42**
UCB average expected regret per round: 4.81

4 – Optimization with uncertain conversion rates, alpha ratios and number of items sold per product

Optimization with **uncertain α**

Binary features cannot be observed and therefore **data are aggregated**

Number of items sold is **UNKNOWN**:
[2, 1, 3, 3, 1]

We provide an estimation and update it with the mean (daily_units[i] / purchases[i])

TS Algorithm Variation

adding support for multiple buys according to a random variable

pull_arm

Pulls the correct arm to be played at the next round...

First exploration of all possible arms

We now consider a random quantity associated with the pulled arm

```
def pull_arm(self, margins_matrix):

  if self.t < self.n_arms:
    return np.array([self.t] * 5)
  idx = np.zeros(5)
  for i in range(5):
    idx[i] = np.argmax(
      np.random.beta(self.beta_parameters[i][:, 0], self.beta_parameters[i][:, 1]) * margins_matrix[i, :]
      * self.lambda_poisson[i])

  return idx
```

__init__

Initializing with extra variables

```
def __init__(self, n_arms):
  super().__init__(n_arms)
  self.beta_parameters = np.array([np.ones((n_arms, 2))] * 5)
  self.expected_rewards = np.zeros([5, n_arms])
  self.lambda_poisson = np.array([np.zeros(4)] * 5)
```

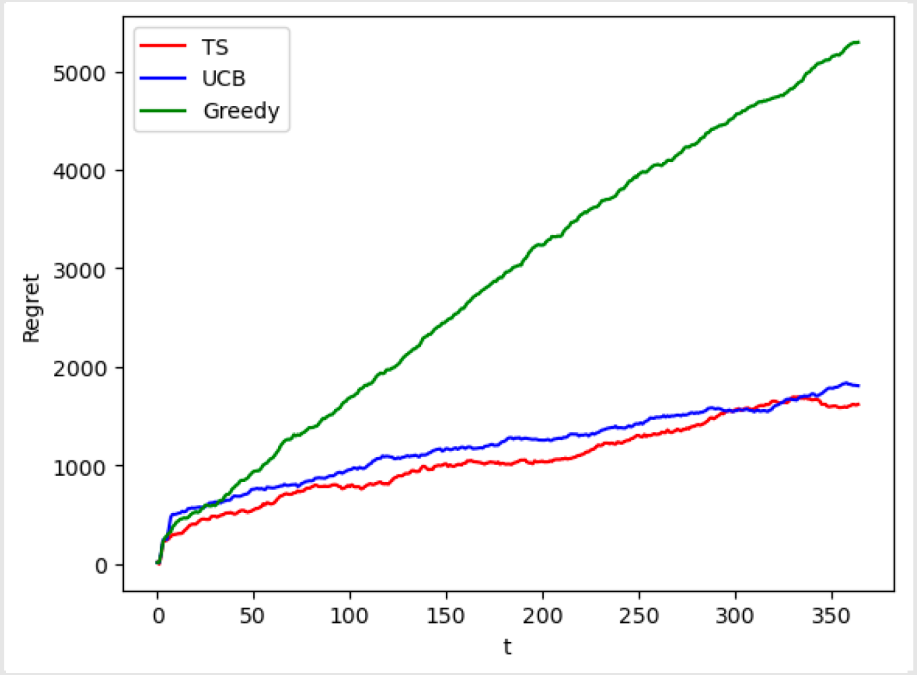
update

The update function is modified in order to handle the purchases of multiple item

```
def update(self, pulled_arm, reward, clicks, purchases, daily_units):
  self.t += 1
  self.update_observations(pulled_arm, reward)
  for i in range(5):
    self.beta_parameters[i][int(pulled_arm[i]), 0] = self.beta_parameters[i][int(pulled_arm[i]), 0] + purchases[i]
    self.beta_parameters[i][int(pulled_arm[i]), 1] = self.beta_parameters[i][int(pulled_arm[i]), 1] + (clicks[i] - purchases[i])
    self.counter_per_arm[i][int(pulled_arm[i])] += 1
    self.expected_rewards[i][int(pulled_arm[i])] = (self.expected_rewards[i][int(pulled_arm[i])] * (
      self.counter_per_arm[i][int(pulled_arm[i])] - 1) + reward[i]) / self.counter_per_arm[i][int(pulled_arm[i])]
  if purchases[i] != 0:
    self.lambda_poisson[i][int(pulled_arm[i])] = (self.lambda_poisson[i][int(pulled_arm[i])] * (
      self.counter_per_arm[i][int(pulled_arm[i])] - 1) + daily_units[i] / purchases[i]) / \
      self.counter_per_arm[i][int(pulled_arm[i])] if \
    self.lambda_poisson[i][int(pulled_arm[i])] > 0 else daily_units[i] / purchases[i]
```

Learning Algorithms Performance

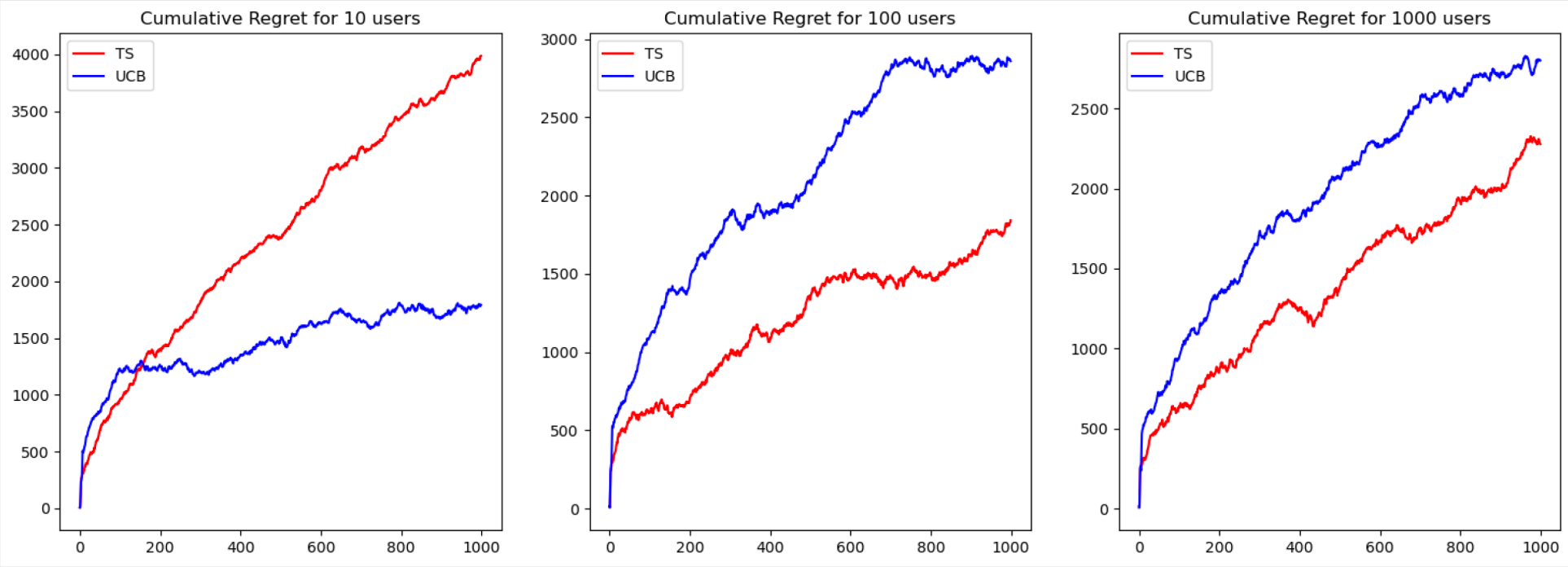
Cumulative regret over time



Learning Algorithms Performance

Cumulative regret over time depending on the number of users

10, 100, 1000 users



Results step 4

Results considering a period of 365 days over 20 runs:

Expected total collected rewards with the best configuration: **48804**

TS Total collected reward: **47189** with 904 of standard deviation between the experiments
UCB Total collected reward: 47000 with 687 of standard deviation between the experiments

TS cumulative expected regret: **1615**
UCB cumulative expected regret: 1804

Expected reward per round with the best configuration: **133.71**

TS average expected reward per round: **129.28**
UCB average expected reward per round: 128.76

TS average expected regret per round: **4.43**
UCB average expected regret per round: 4.94

5 – Optimization with uncertain graph weights

Optimization with **uncertain α**

Binary features cannot be observed and therefore **data are aggregated**

Number of items sold is **known:**
[2, 1, 3, 3, 1]

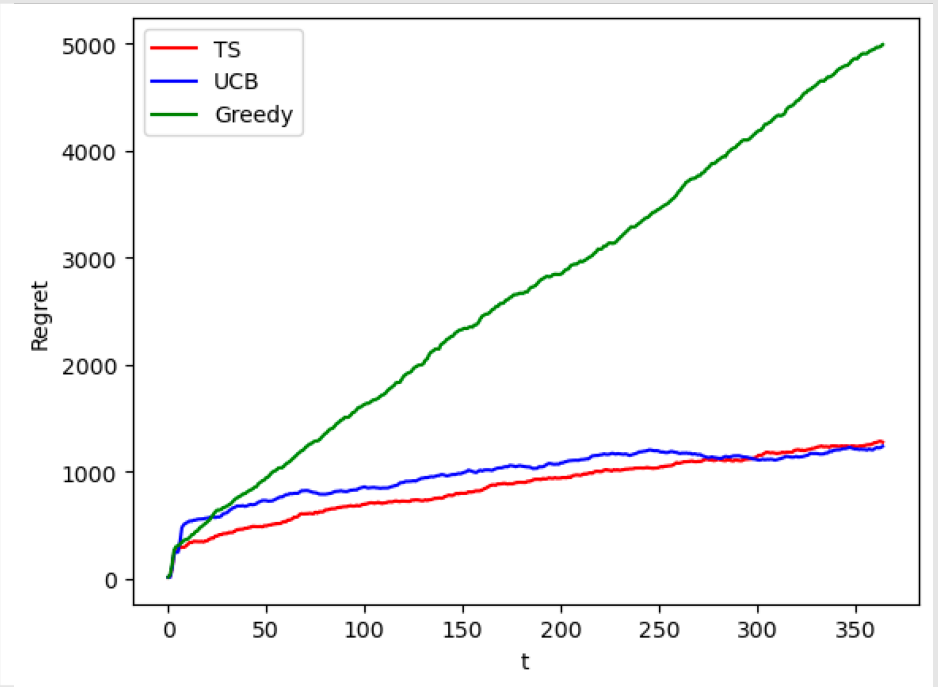
Algorithm Variation

The random weights are defined inside the User class

```
self.P = User.generate_graph(self, np.random.uniform(0.2, 1, size=(5, 5)))
```

Learning Algorithms Performance

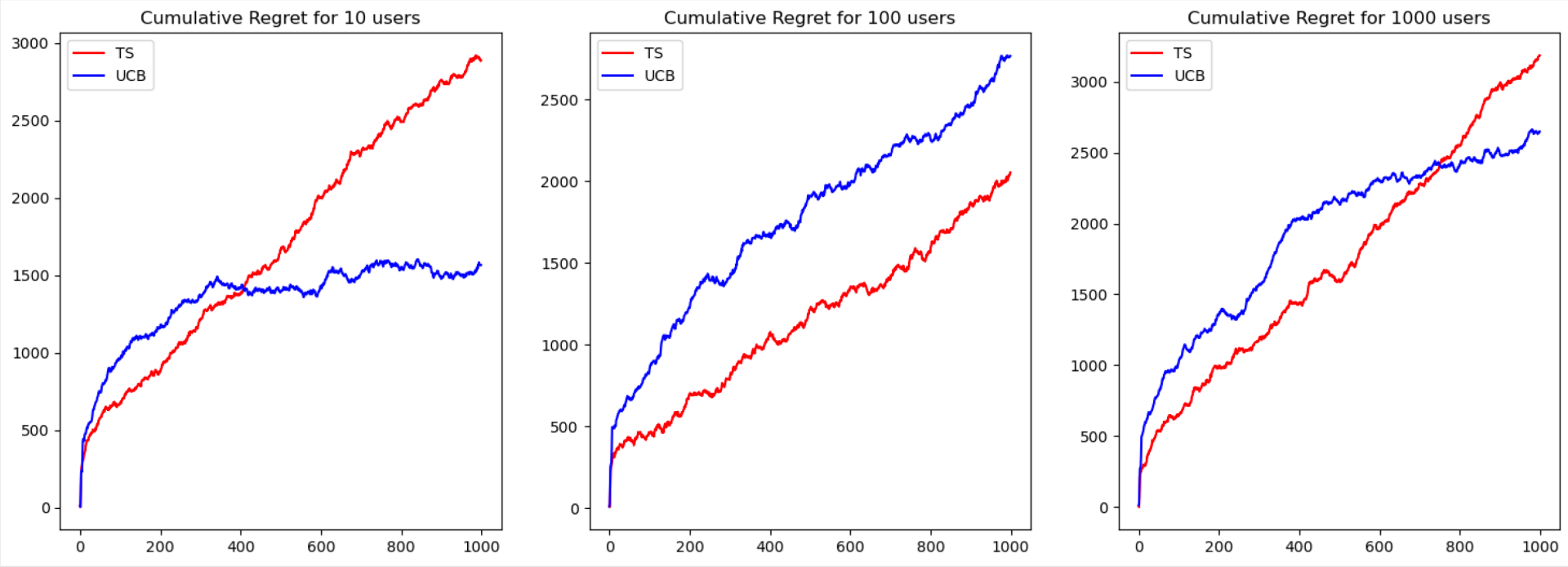
Cumulative regret over time



Learning Algorithms Performance

Cumulative regret over time depending on the number of users

10, 100, 1000 users



Results step 5

Results considering a period of 365 days over 20 runs:

Expected total collected rewards with the best configuration: 48873

TS Total collected reward: 47602 with 863 of standard deviation between the experiments
UCB Total collected reward: **47641** with 547 of standard deviation between the experiments

TS cumulative expected regret: 1271
UCB cumulative expected regret: **1232**

Expected reward per round with the best configuration: 133.90

TS average expected reward per round: 130.41
UCB average expected reward per round: **130.52**

TS average expected regret per round: 3.48
UCB average expected regret per round: **3.38**

Step 6 - Non stationary demand curves

The reservation price can **change abruptly**

ASSUMPTION:
2 changes per year based on the season (fall/winter, spring/summer)

Change = 1.15, 0.95, 1.2, 0.9, 1

UCB Algorithm Variations

SW UCB

Sliding-window UCB algorithm.

param

n_arms: number of prices
window_size: size of the window (constant)

CUSUM change detection

param

n_arms: number of arms
M: initialization of CUSUM
eps: minimum expected mean variation
h: threshold for the CUSUM walk
c: constant of UCB

Algorithm 2 Two-sided CUSUM

Fix parameters ε, M, $\{y_k\}$. k≥1

Initialize $g_0^+ = 0$ and $g_0^- = 0$.

For each k do

Calculate s_k^+ and s_k^- according to (6).
Update g_k^+ and g_k^- according to (7).
if $g_k^+ \geq h$ or $g_k^- \geq h$
return 1
end if

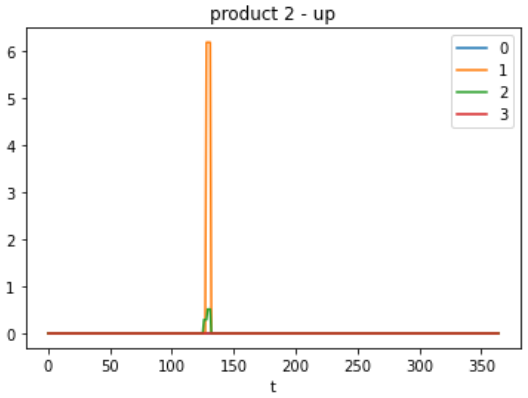
end for

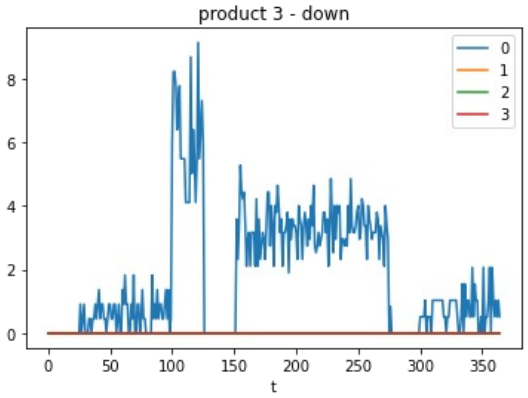
$$(s_k^+, s_k^-) = (y_k - \hat{u}_0 - \epsilon, \hat{u}_0 - y_k - \epsilon)\mathbb{1}_{\{k>M\}}. \quad (6)$$

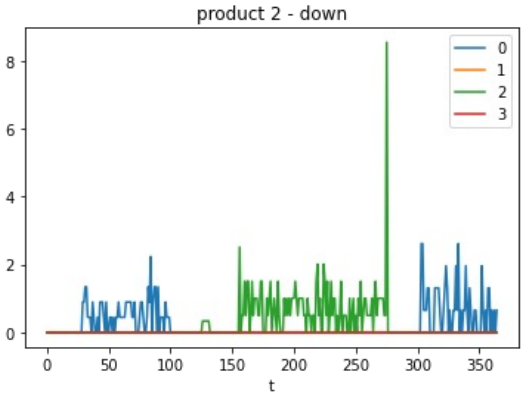
Let g_k^+ (g_k^-) track the positive drift of upper (lower) random walk. In particular,

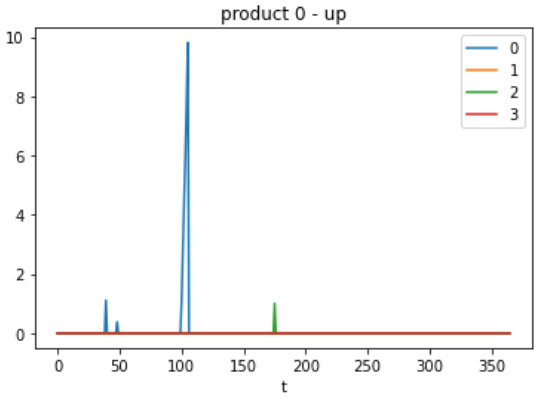
$$g_k^+ = \max(0, g_{k-1}^+ + s_k^+), \; g_k^- = \max(0, g_{k-1}^- + s_k^-). \quad (7)$$

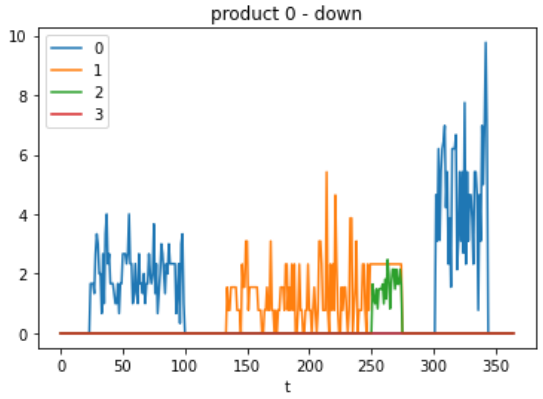
CUSUM ALGORITHM – threshold definition











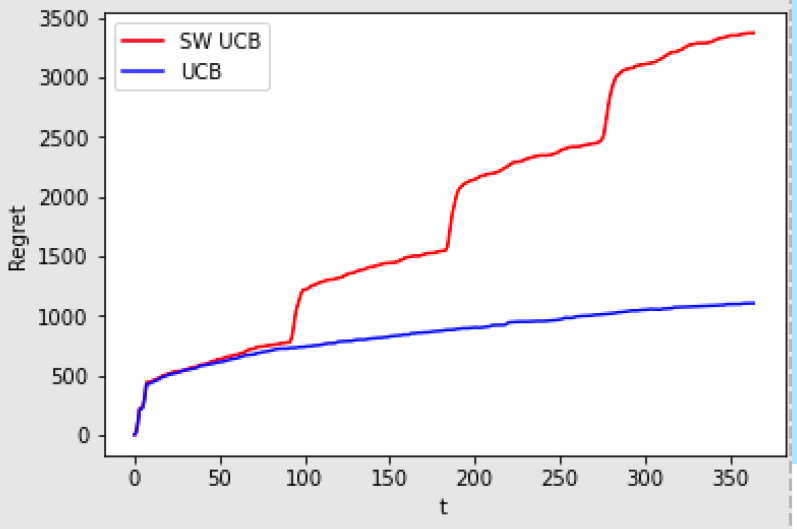
ε is set a priori based on sample variance of the conversion rates

the threshold is set by trying to avoid being unnecessarily triggered

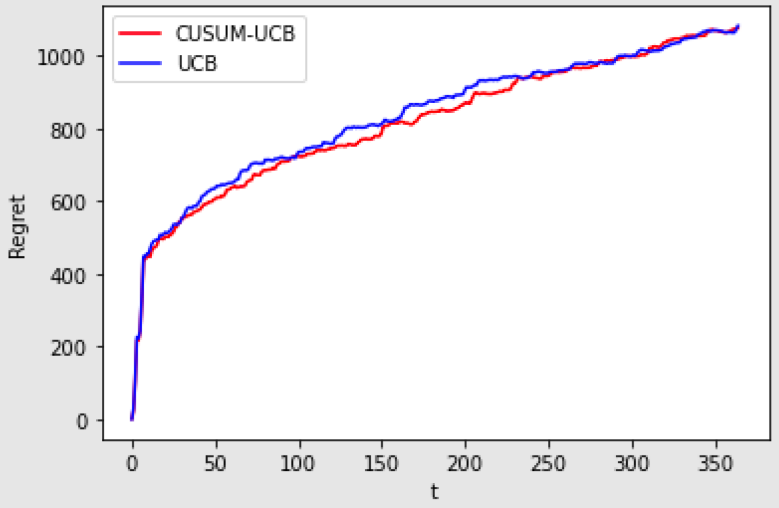
Algorithm Variations PERFORMANCE

STATIC

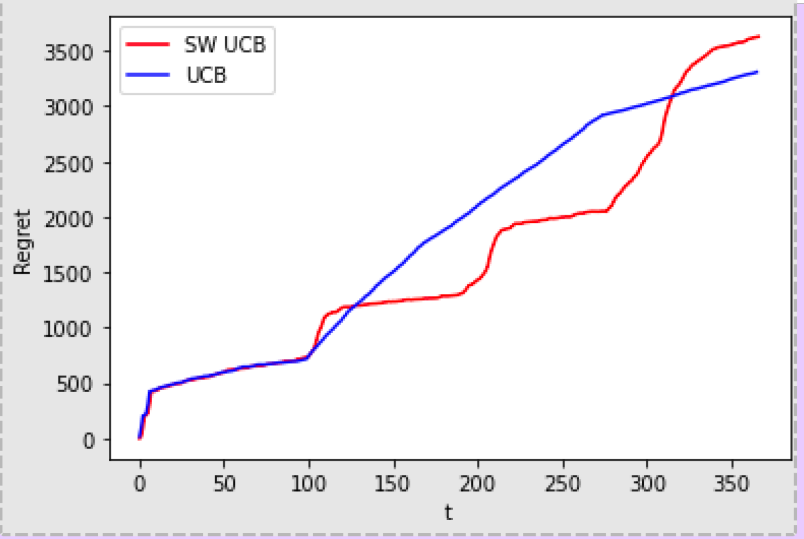
SW UCB

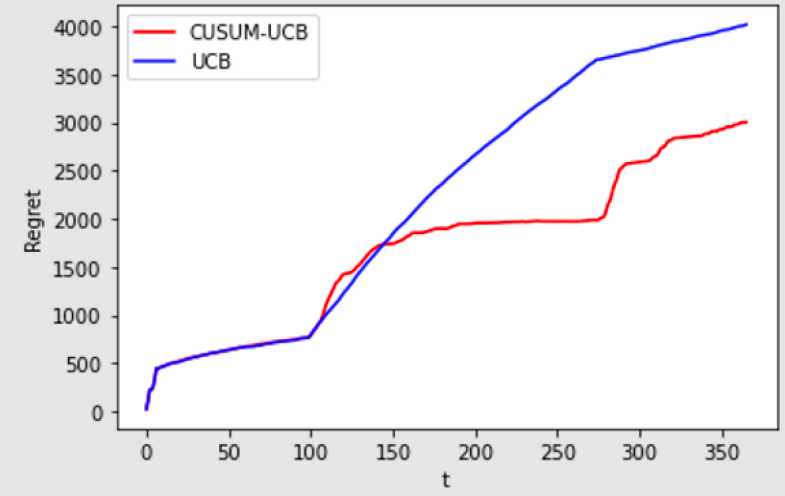


CUMSUM UCB



DYNAMIC





RESULTS STEP 6 - Static SW

Total expected reward over 365 days and its standard deviation considering 20 experiments

SW-UCB 40007 138

CUSUM-UCB 42197 81

Total expected regret over 365 days and its standard deviation considering 20 experiments

SW-UCB 3300

CUSUM-UCB 1109

Average expected reward per round over 365 days and its standard deviation between the rounds considering 20 experiments

SW-UCB 110

CUSUM-UCB 115

Average expected regret per round over 365 days and its standard deviation between the rounds considering 20 experiments

SW-UCB 9.04

UCB CUSUM-UCB 3.04

Dynamic SW

Total expected reward over 365 days and its standard deviation considering 10 experiments

SW-UCB 40018 156

CUSUM-UCB 40800 806

Total expected regret over 365 days and its standard deviation considering 10 experiments

SW-UCB 3810

CUSUM-UCB 3045

Average expected reward per round over 365 days and its standard deviation between the rounds considering 10 experiments

SW-UCB 110

CUSUM-UCB 112

Average expected regret per round over 365 days and its standard deviation between the rounds considering 10 experiments

SW-UCB 10.90

CUSUM-UCB 8.61

Step 7 – Context generation

We assume that we can record all the features on the website and that we can set a different price for each user's class [position-based prices and discounts for the premium members)

New E-commerce structure: we consider each context separately

Every 14 days, we test if splitting a feature is worth it or not through the condition:

Step 7 – Context generation

We assume that we can record all the features on the website and that we can set a **different prices for each user class**

→ Location-based prices and discounts for the premium members

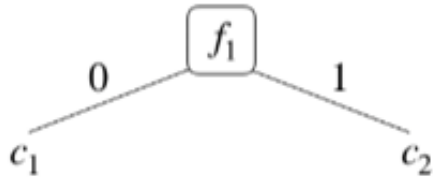
New E-commerce structure:
We consider each context separately

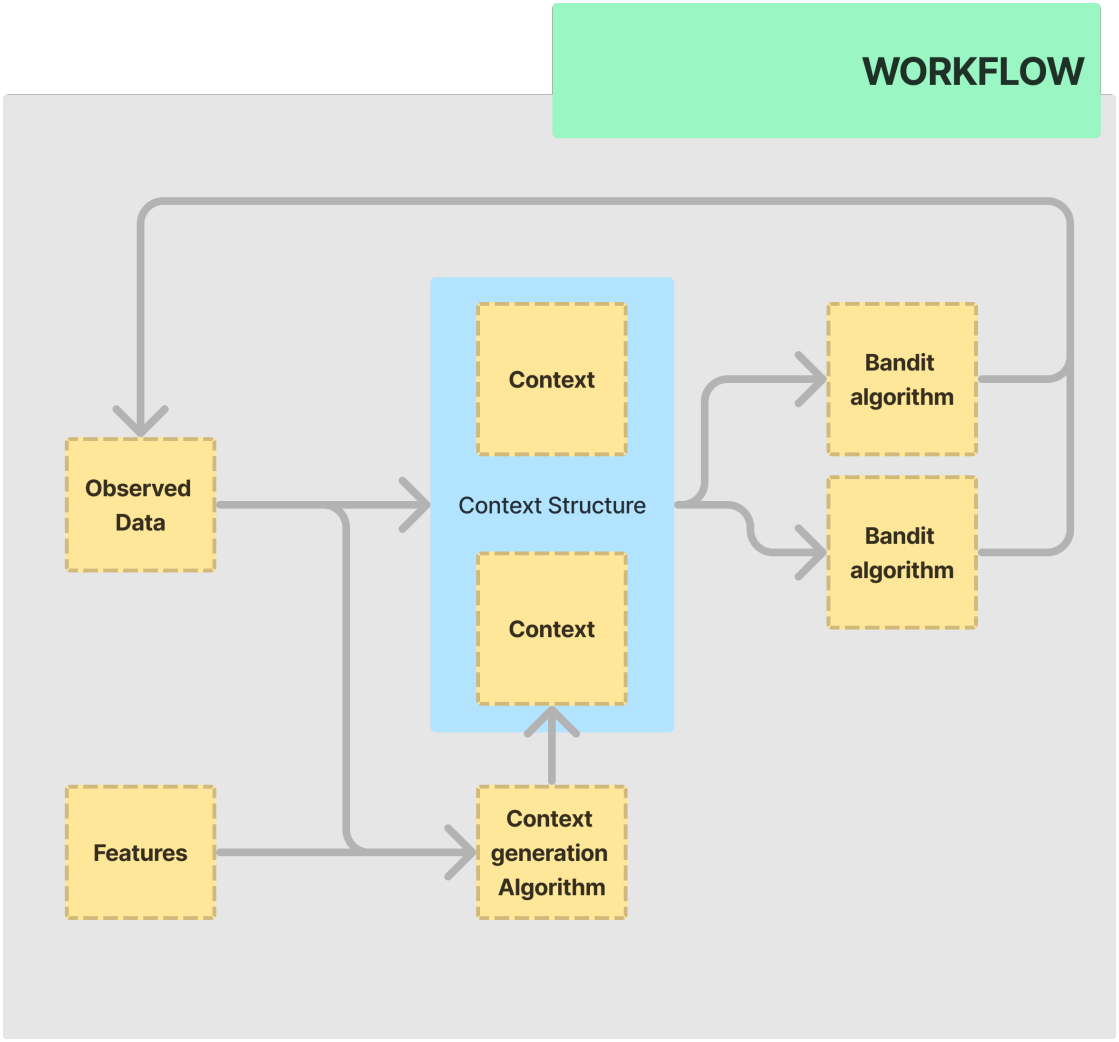
Every 14 days
We test if splitting a feature is worth it or not

Split Condition

- Based on

$$p_{c_1} * \mu_{c_1} + p_{c_2} * \mu_{c_2} \geq \mu_{c_0}$$





Simulation of the context generation for TS

Period 0
mu_c0_ts= [[1185.7462826575595, 0], [0, 0]]

period 1
mu_c0_ts= [[1185.7462826575595, 0], [1928.443288310946, 0]]
p_c1*mu_c1 + p_c2*mu_c2= 1928.443288310946
--- SPLIT F1 MAKES SENSE ---

period 2
mu_c0_ts= [[1185.7462826575595, 0], [1928.443288310946, 1903.8843444854774]]
p_c1*mu_c1 + p_c2*mu_c2= 1903.8843444854774
--- SPLIT F2 DOESN'T MAKE SENSE (we consider only f1)---

period 3
mu_c0_ts= [[1185.7462826575595, 0], [1864.4348796352126, 1903.8843444854774]]
p_c1*mu_c1 + p_c2*mu_c2= 1864.4348796352126
--- SPLIT F1 MAKES SENSE ---

Period 4
mu_c0_ts= [[1185.7462826575595, 0], [1864.4348796352126, 1940.9036865966405]]
p_c1*mu_c1 + p_c2*mu_c2= 1940.9036865966405
--- SPLIT F2 MAKES SENSE (together with f1)---

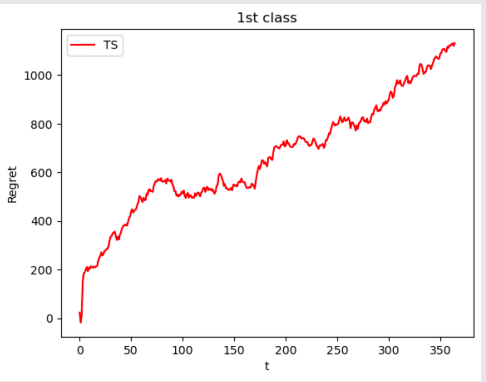
Period 5
mu_c0_ts= [[1185.7462826575595, 0], [1864.4348796352126, 2313.224688074132]]
p_c1*mu_c1 + p_c2*mu_c2= 2313.224688074132
SPLIT F2 MAKES SENSE (together with f1)---

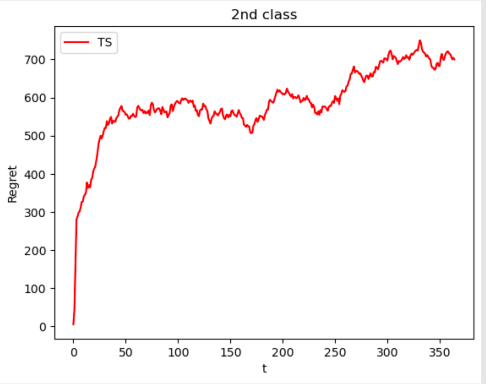
1st split scenario

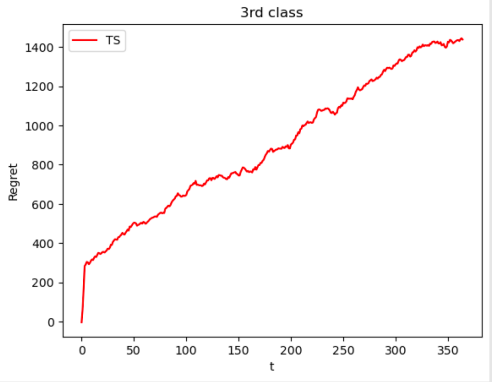
2 classes of users

ts_2_classes_rewards =
0.3*ts_class1 + 0.7 * ts_class2

2nd split scenario

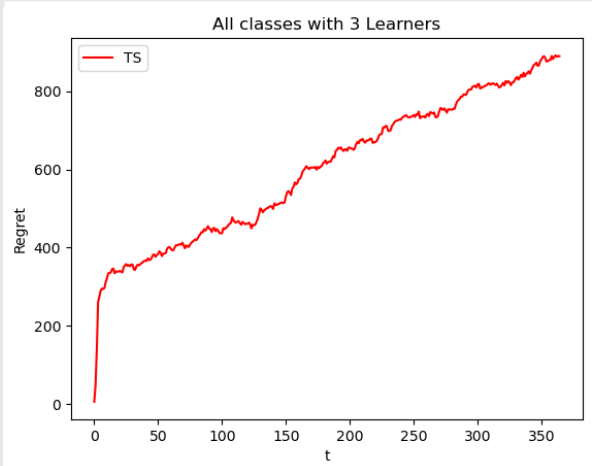




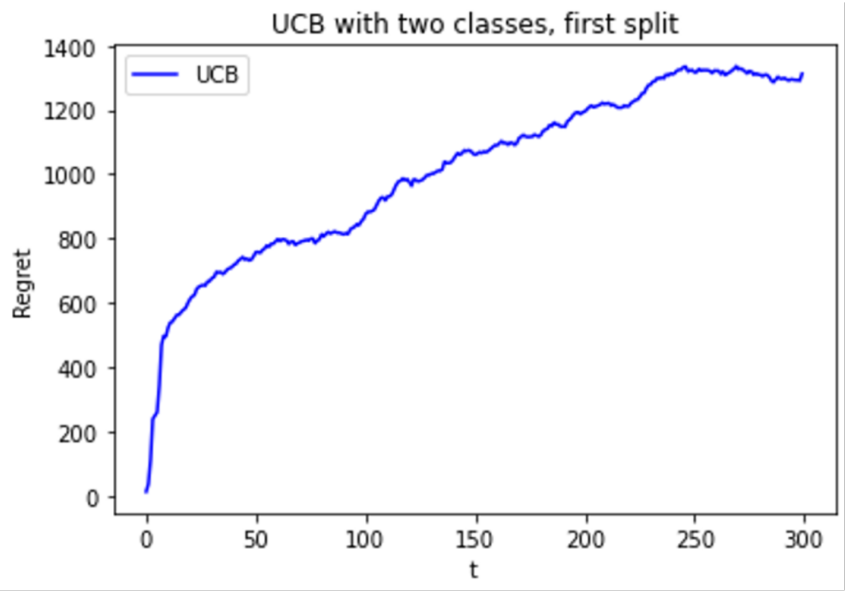


3 classes of users

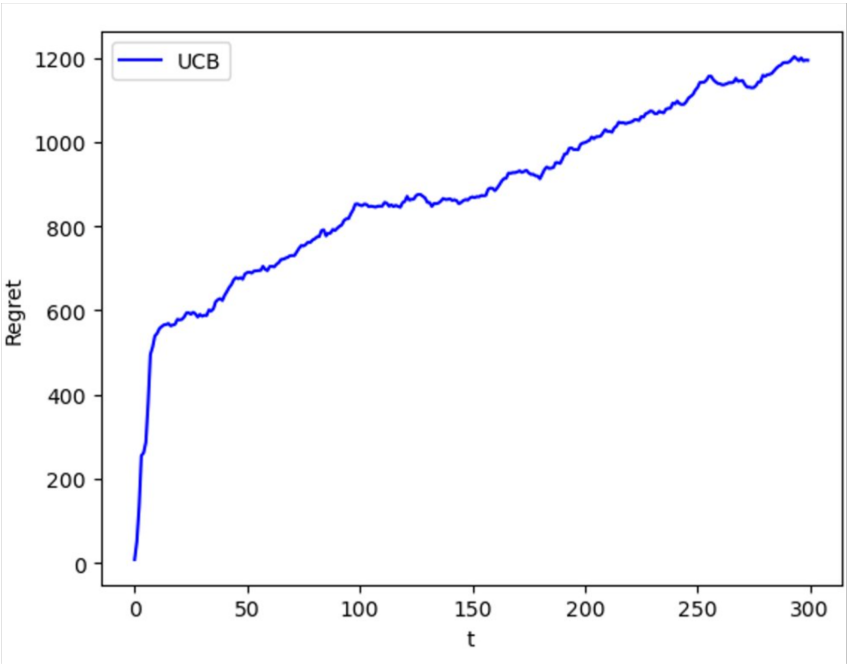
ts_all_classes_rewards =
0.3*ts_class1 +
0.4 * ts_class2 +
0.3 * ts_class3



UCB in the first split scenario
2 classes of users



UCB in the second split scenario
3 classes of users



Splitting in 3 classes instead of 2 only make sense over a long enough time horizon

Over 14 days, we can't see a big enough improvement

Results step 7

Considering the optimal solution per round: **125.66**

Thomson Sampling

Average expected reward per round on 365 days over 10 runs: **122.81**

Average expected regret per round in 365 days over 10 runs : **2.85**

Average total regret in 365 days over 10 runs: **1040**

Average total reward in 365 days over 10 runs: 44828 with a standard deviation of 1484 between the experiments

UCB

Average expected reward per round on 365 days over 10 runs: 121.68

Average expected regret per round in 365 days over 10 runs : 3.97

Average total regret in 365 days over 10 runs: 1193

Average total reward in 365 days over 10 runs: **36506** with a standard deviation of 423 between the experiments

Conclusion

Thompson Sampling and UCB are efficient algorithm to deal with pricing optimization problems

Competent under uncertainty

In general out-perform random choices and greedy algorithms

Split model being superior

Resilient in not stationary condition (for little variation)

Thank You

Images from Pau Barbaro on blush.design
Images by pch.vector on Freepik