# Reinforcement Learning : Assignment 1

DIRHOUSSI Ahmed Amine

March 15, 2020

**Abstract**

In the following we will review the Cascading Bandits paper [2].We will provide an overview of the problem it tries to solve, the methodology and algorithms described in the paper as well as the results obtained. We will also implement one of the algorithms described in the paper and review the main results in light of the literature.

# 1 Summary

## 1.1 Main problem

Ranking and presenting web pages by attractiveness to user is very important problem. The search engine has to 'learn' the user behavior to output the most attractive items to a user from a set of web pages, where attractiveness can usually be inferred from historical click data of the user. The paper presents a 'novel' way of modeling this interaction in an online setting based on the **Cascade Model**.

We could formulate a simpler setup where the search engine has to recommend the most attractive item. This could easily be modeled as a stochastic bandit problem where each page/item has an unknown attractiveness probability. This problem can be solved with classic stochastic algorithms like **UCB1**. The main difference here is the combinatorial aspect of the problem where we need to recommend $K$ items from $L$. The model is related to a form of bandits problem where each recommendation is modeled as a separate independent bandit. We call this type of modeling **Ranked bandits**, we will discuss the differences between **Ranked bandits** and **Cascade bandit** will be discussed in section 3.

## 1.2 Cascade Bandit

The **cascade model** is a very useful framework to model the user behavior in web page recommendation. Before diving into the main results presented in the paper we will setup the different notation used in describing the **cascading bandit** presented in paper.

The user is presented at each round $t$ with a list of $K$ items $A_t = \{a_1^t, .., a_K^t\}$ from a set $E = \{1, .., L\}$ of items for $t \in 1, .., n$. The user will then examine this list and click on the attractive item, let say item $e$ in the list. This choice divides the list of items into two distinct separate groups :

1. **Observed items**: All items before $e$. These items are not attractive for the user in that round.
2. **Unobserved items**: All items coming after $e$, these are not observed as the user stops examining the list after clicking on item $e$.

The paper models the user behavior as an unknown probability distribution of attractiveness on the items $P$ : a probability distribution over $\{0, 1\}^E$. The user samples a sequence of weights $(w_t)_{t \in \{1,..,n\}}$ with $w_t \in \{0, 1\}^E$, the choices are then randomly sampled and are not adjusted whilst interacting with the system. In this setting, at time $t$, the user will click on the first item $e$ where $w_t(e) = 1$. The model does not support multiple clicks on the item, we will discuss this limitation later in section 3.

The paper defines the reward at $t$ as a proxy to the probability that at least one item of the list $A_t$ is attractive to the user :

$$f(A_t, w_t) = 1 - \prod_{k=1}^{K} (1 - w_t(a_k)) \tag{1}$$

This reward is in a sense oblivious to the quality of the ranking, we could have imagined a world where we assume that poorly ranked attractive item gives a lower reward because the user had to scroll down a long list of unattractive items before finding it. This reward is non-linear in the unknown parameters $w(a_k)$.
Given what was said before, the feedback observed at round $t$ is the index of the first attractive item : $C_t$ ($\infty$ if no item was clicked on). We could then infer the weights of **observed items** at that round $w_t(e) = \mathbb{1}(C_t = e)$ for $1 \leq min(C_t, K)$. Two necessary assumptions were made to define the **cascading bandits** and further the analysis :

1. The weights of the items is independently distributed. At each time t, the weight of item $e$ is drawn independently of the weights of other items in any $t$.
2. The weight of item $e$ in $E$ follows a Bernoulli distribution parameterized by $\bar{w}(e)$

These assumption were essential to link the reward to the attraction probability of the items in $A$ :

$$\mathbb{E}[f(A, w)] = f(A, \bar{w}) \tag{2}$$

This expected reward is maximized by the K items with the highest probability of being attractive.
The paper also defines the **expected cumulative reward** to evaluate the learner's policy as :

$$R(n) = \mathbb{E}[\sum_{t=1}^{n} R(A_t, w_t)] \tag{3}$$

with $R(A_t, w_t) = f(A^*, w_t) - f(A_t, w_t)$, $A^*$ being the optimal list of items to present i.e. the list that yields the highest expected reward.

## 1.3 Algorithms

Two algorithms were proposed to solve the problem described above. They are both based on an optimistic policy where the algorithm estimates upper confidence bounds of the $\bar{w}(e)$ and shortlists the first $K$ items with the highest estimated $UCB$. The optimism principle means using the rewards observed for each item to compute an overestimate (with high probability) of the unknown weight. This means that displaying sub-optimal item will only happen if they have a high $UCB$ which cannot happen too often because of the additional data provided when displaying those. This means that the users will probably not click on them and their $UCB$ will eventually fall below the optimal items.

The two algorithms follow the structure presented in 1. The `CascadeUCB1` and `CascadeKL-UCB`, only differ in the way they compute the $U_t$. The confidence bounds of `CascadeUCB1` are derived from the Hoeffding's inequality by defining a radius of confidence $c_{t,s} = \sqrt{1.5log(t)/s}$, where s is the number of time the item is observed. `CascadeKL-UCB` uses the Chernoff's lemma and the Kullback-Leibler divergence to derive an upper estimate of the weight. This computation is marginally costlier compared to the `CascadeUCB1` but the improvement is quite significant especially when most items have a low probability of being attractive.

## 1.4 Main results

### 1.4.1 Regret analysis

The first result presented in the paper is an gap dependent upper bound on the $n$-step regret of `CascadeUCB1` and `CascadeKL-UCB`. The gap is defined as $\Delta_{e,e^*} = \bar{w}(e^*) - \bar{w}(e)$ and represents the difference in probability of attraction of an item $e$ to an optimal item $e^*$. In the following we will consider a constant gap for all suboptimal items : $\Delta$, and the same probability of being attractive for optimal items, $p$. Using the fact that

**Algorithm 1:** Cascading bandit algorithm structure

**Step 0 : Initialization**
**while** $t \leq L$ **do**
  Recommend each item $e$ as the first and randomly choose the $k-1$ other
**end while**
**while** $t \leq n$ **do**
  **Step 1 : Select a partition of items based on sorting rule**
  Compute UCBs of items : $U_t$
  Output K items: $A_t \leftarrow (a_1^t, ..., a_K^t)$ based of the sorting of $U_t(e)$
  **Step 2 : Update the sorting procedure data**
  Get feedback $C_t \in \{1, ..., K, \infty\}$
  Compute the number of time we observed item $e$
  Update empirical weight estimate for observed item $e$
  $i \leftarrow i - 1$
**end while**

the CascadeUCB algorithm only suffers regret if it recommends sub-optimal that are *observed*, the paper proves upper bounds for `CascadeUCB1` in :

$$O((L-K)log(n)\frac{1}{\Delta}) = O((L-K)log(n)\frac{K\Delta}{D_{KL}(p-\Delta||p)}) \tag{4}$$

A similar upper bound is proved for `CascadeKL-UCB` :

$$O((L-K)log(n)\frac{\Delta(1+log(1/\Delta)}{D_{KL}(p-\Delta||p)}) \tag{5}$$

where $D_{KL}$ is the *Kullback-Leibler divergence*.

These bounds are logarithmic in the number of rounds, linear in the number of items $L$ (unpleasant when $L$ is large) but improve as the number of recommended item $K$ increases. We note that the upper bound `CascadeKL-UCB` is better when the gap $\Delta = \Omega(e^{-K})$. The lower bounds proved by for the two algorithm match the upper bound up to a factor $log(1/\Delta)$, where the lower bound proved is :

$$O((L-K)log(n)\frac{\Delta}{D_{KL}(p-\Delta||p)}) \tag{6}$$

### 1.4.2 Experiments

**Validating regret bounds :** Running the `CascadeUCB1` and `CascadeKL-UCB` in $10^5$ steps. The main results were that the regret scales with the number of items $L$ and decrease when $K$ increases which validates the $O(L-K)$ bound. The regret increases also when the gap $\Delta$ decreases. We also note that `CascadeKL-UCB` has better regret in the experiments which is explainable by the fact that the confidence interval of our weight estimate gets narrower when $p$ is close to 0 or 1.

**Experimenting with inverse ordering of recommended items :** This was a very interesting experiment. Changing the order of recommendation (recommending lower UCB items in $A_t$ first did not affect the total regret. This is explainable by the fact that the regret modeled does not take into account the ranking of the attractive item.

**Testing robustness in face of violated assumptions :** The `CascadeKL-UCB` algorithm was tested in another setting where the previous 'simple' modeling assumptions were violated. In this extension of the previous model, the paper introduces a novel way of modeling the weight of items as a product of attraction and satisfaction : $\bar{w}(e) = \rho(e)\nu(e)$. The user can click on multiple items and jump from an item to another

with a persistence probability $\gamma$. The paper tested the `CascadeKL-UCB` with two fixed values of $\nu(e)$ for all items and two fixed persistence $\gamma$. The experiments takes the last user click as a signal of satisfaction. The `CascadeKL-UCB` performed surprisingly well and converged to a solution in all the settings.

**Comparison to ranked bandits :** Following the last experiment, the paper compared the `CascadeKL-UCB` results to `RankedKL-UCB` algorithm. The regret of the latter algorithm was significantly higher than that of `CascadeKL-UCB`. This is explained by the fact that regret scales in $O(L-K)$ for `CascadeKL-UCB` (see 5) and $O(KL)$ for the `RankedKL-UCB` presented by Radlinski et al. [4].

## 2 Implementation of CascadeUCB

To test the result presented by the paper, I implemented the algorithm `CascadeUCB` described above in algorithm 1.3. The $UCB$ computation was based on `CascadeUCB1`. The implementation code details can be found in appendix A. Running the code for $L = 8$, $K = 2$, $\Delta = 0.15$, $p = 0.2$ and for $n = 10^5$ rounds we clearly see a convergence as the cumulative regret stabilizes.

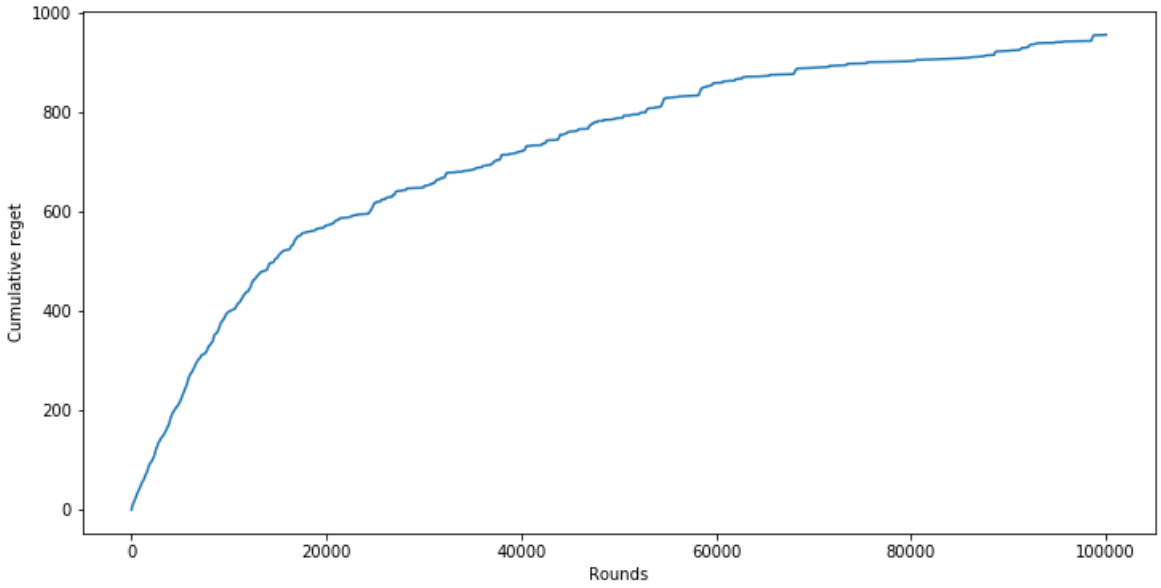


Figure 1: Convergence for $L = 8$, $K = 2$, $\Delta = 0.15$, $p = 0.2$ and for $n = 10^5$

Plotting the selected item $e$ by rounds on a logarithmic scale on the $y$-axis in figure 2, we clearly see that the algorithms stops selecting sub-optimal items (items from 2 to 8).

To validate the regret bounds proved in section 1.4.1, i ran an experiment with 4 different combinations of values for $K$,$L$ and $\Delta$, the average cumulative regret and standard deviation after 5 runs are presented in the table 1.

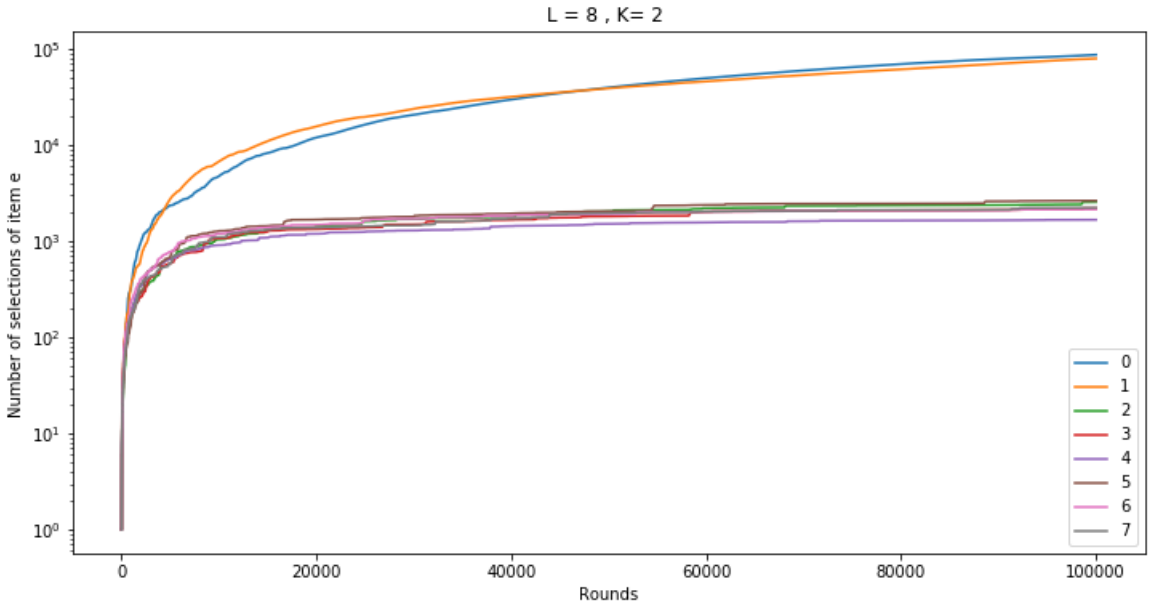


Figure 2: $L = 8$, $K = 2$, $\Delta = 0.15$, $p = 0.2$ and for $n = 10^5$

| | L | K | $\Delta$ | cumul.regret $\pm$std |
|---|---|---|---|---|
| 0 | 16 | 2 | 0.15 | 1450.31$\pm$104.0 |
| 1 | 16 | 4 | 0.15 | 1126.88$\pm$87.18 |
| 2 | 16 | 4 | 0.075 | 1676.82$\pm$74.85 |
| 3 | 8 | 2 | 0.075 | 981.87$\pm$31.37 |

Table 1: Cumulative regret after $n = 10^5$ rounds and 5 trials

We do in fact retrieve the upper bounds of the regret in equation 4. The regret increases linearly in $L - K$. When doubling the number of items $L$, the regret doubles. Increasing $K$ with a constant number of items shows that the regret decreases. The regret also decreases when $\Delta$ increases, confirming the $1/\Delta$ factor in the upper bound.

# 3 Literature review

Choosing a ranked list of $K$ items even under stochastic assumptions on the user clicks is a very difficult problem. To illustrate this claim, let's consider the case where the user can click on multiple items, and let $C_{t,i} \in \{0, 1\}$ be the observed feedback at time $t$ for item $i$ ($C_{t,i} = 1$ if item clicked on the $ith$ item). Choosing $K$ items from a list of $L$ to maximize number of clicks can be framed as a finite-armed bandit problem where each arm is the possible ranking of $K$ items. This problem is intractable in reality because the way to rank $K$ items from $L$ is of size $L!/(L - K)!$, which is still quite large given that the regret of algorithms like `UCB1` scales with the square root of the number of arms. In the context of ranking, we have to make some assumption on the value function on ranking the items. An exhaustive list of these assumptions on user behavior known as click models can be found in the survey paper by Chuklin et al. [1]. Some of the most popular assumption choices are :

1. **Document Based Model :** In the document-based model, the probability of clicking on item $e$ is

equal to its **attractiveness**, plain and simple. The unknown quantity in this model is the attractiveness function which maps the $L$ items to a value in $[0, 1]$.

2. **Position Based Model :** In this model, we assume that the position of the item affects the likelihood of the click. We define a function $\chi : \{1, ...K\} \to [0, 1]$ which measures the quality of the position. The overall value of item $e$ positioned at potion $k$ is equal to the product of its attractiveness and the quality of the position. This model is richer than the **document-based model** but we'll have to learn $L + K$ parameters.

3. **Cascade Model :** This is the model presented in the paper. We assume that the user clicks on *the first item* that attracts him. The probability of clicking on an item $k$ is then modeled as a product of the intrinsic attractiveness of the item times the probability of not finding the $k - 1$ items before attractive (assuming that each item attracts the user independently).

4. **Dependent Click Model :** While the cascade model assumes that the users stops examining the list of item after clicking on the first attractive item, this assumption is too restrictive in real-world application. Guo et al. [3] introduced the **Dependent Click Model**, a multiple-click generalization of the cascade model. In this model the user examines the list of item, clicks on item $a_k$ with attraction probability $w(a_k)$ (independent of other items). The user terminate the search then with probability $v(k)$. In this case the user is satisfied. The reward function in this setup ( probability of user leaving satisfied) is very close to the one in the cascading bandit setup : $f(A, w, v) = 1 - \prod_{k=1}^{K}(1 - v(k)w(a_k))$.

The fourth experiment presented in the paper (section 1.4.2) compares the `CascadeKL-UCB` algorithm to `RankedKL-UCB`. The Ranked bandit algorithm was first introduced by by Radlinski et al. [4]. The main idea in this algorithm is to model each of the $K$ positions as a separate bandit problem and solve it using a *base* bandit algorithm. In the ranked bandit algorithm, the first arm (MAB responsible for selecting item in rank 1) chooses item $i$ in $[L]$. The the second arm chooses an item $j$. We then check if the second item is the same as the first, if so we replace it by a random item. In general, the ranked bandit algorithm learn a $1 - 1/e$ approximate solution of the maximum coverage problem (optimal $K$ from $L$ documents for a given user) but the regret associate scales in $O(KL)$.

Combes et al., [6] proposed a different algorithm around the same time for the same click model. In this paper, we assume that items are categorized by topic and users are clustered. We assume that users from the same cluster are interested in the same topic. The authors proposed `PIE` (Parsimonious Item Exploration) algorithm when the topic and the user is known, and `PIE-C` where only the user cluster is known. The lower bound for both algorithm are asymptotically optimal and are very similar to those presented on this paper 5.

The main limitation of the cascading bandits is that they cannot learn from multiple clicks. A generalization is presented in by Karaiya et al. [8] paper. The paper studies the **dependent click model** which differs from the model proposed in this paper by allowing the user to click on multiple items and the reward not assumed to be observed. This is a key difference between the cascading bandits problem where the user can click on at most one item and this click is satisfactory. The key idea presented to learn from this setting is to assume that the termination probabilities have a **known order**. This framework naturally induces the necessity to rank items in the optimal order. The algorithm presented by Karaiya et al. [8], `dcmKL-UCB` is very close to the `CascadeKL-UCB`, except the addition of the weight probabilities. We retrieve basically matching upper and lower regret bounds those of CascadeKL-UCB.

In the cascade Model, the positions of the items are not taken into account in the reward because it assumes to obtain a click as long as the interesting item belongs to the list proposed. This is clearly highlighted in the paper when experimenting with a reverse ordering of item, see section 1.4.2. To overcome these limitations, the position based model was studied by Lagree et al. [5]. In this setting, we add a binary unknown *Examination variable* corresponding to the user actually examining the item in the list. This problem is studied then as a stochastic multiple-play bandits with semi-bandit feedback.

In the all previous algorithms we have a linear dependence in $L$ the number of items. This could be problematic in practice when $L$ is very large, for instance choosing 10 movies from a ground set 100k movies. To overcome this problem, we look at a linear variant where the attractiveness of an item is

assumed to be an inner product between a feature vector of the item and an unknown parameter $\bar{w}(e) = x(e)^T\theta*$. This modeling assumption was studied by Zong et al. [7]. The paper suggested two algorithms for solving this problem `CascadeLinTS` and `CascadeLinUCB` based on the ideas of Thompson sampling and linear UCB. Under the perfect linear generalization assumption, the paper proofs the regret of `CascadeLinUCB` is independent of $L$ and asymptotically optimal.

Finally, Latimore et al.[9] proposed a more generalized click model and a novel learning algorithm `TopRank`. The model poses the problem of finding the $K$ most attractive list as a sorting problem with noisy feedback. Given some general assumptions detailed in the paper, the algorithm proposed maintains a *topological order* of items in each round. The order is represented by relation $G_t$, a pair of items where $(j, i) \in G_t$ means that the algorithm gathered enough evidence up to round $t$ to be sure that item i is more attractive to the user than item j. The `TopRank` algorithm has an upper bound regret of $O(\sqrt{K^3Lnlog(n)})$. Comparing this algorithm to `CascadeKL-UCB`, we clearly see that the the latter algorithm outperforms `TopRank` when assuming the cascade model. This is due to the fact that `CascadeKL-UCB` heavily exploits the knowledge of the cascade click model structure. In the position model setting, `CascadeKL-UCB` suffers from linear regret whereas `TopRank` outperforms it (after 4 million steps). The bad performance of this algorithm is offset by its robustness to multiple click models and a novel, more generalized framework.

# 4 Conclusion

To sum up, ranking items is a very difficult problem and each assumption we make to find a solution has its merits and its limitations. To illustrate that, I would like to quote a brilliant remark from Radlinski et al. [4] criticizing the assumption of associating the ranking to the attractiveness value of items : "The theoretical model that justifies ranking documents [in this way] is the probabilistic ranking principle [Robertson, 1977]. It suggests that documents should be ranked by their probability of relevance to the query. However, the optimality of such a ranking relies on the assumption that there are no statistical dependencies between the probabilities of relevance among documents– an assumption that is clearly violated in practice. For example, if one document about jaguar cars is not relevant to a user who issues the query jaguar, other car pages become less likely to be relevant".

# References


[1] Maarten de Rijke Aleksandr Chuklin Ilya Markov. "Click Models for Web Search". In: (2015). URL: https://clickmodels.weebly.com/uploads/5/2/2/5/52257029/mc2015-clickmodels.pdf.

[2] Branislav Kveton, Csaba Szepesvari, Zheng Wen, Azin Ashkan. "Cascading Bandits: Learning to Rank in the Cascade Model". In: (May 2015). URL: https://arxiv.org/pdf/1502.02763.pdf.

[3] Yi-Min Wang Fan Guo Chao Liu. "Efficient Multiple-Click Models in Web Search". In: (2009). URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.188.471&rep=rep1&type=pdf.

[4] Thorsten Joachims Filip Radlinski Robert Kleinberg. "Learning Diverse Rankings with Multi-Armed Bandits". In: (2008). URL: http://www.cs.cornell.edu/people/tj/publications/radlinski_etal_08a.pdf.

[5] Olivier Cappé Paul Lagrée Claire Vernade. "Multiple-Play Bandits in the Position-Based Model". In: (2016). URL: https://arxiv.org/pdf/1606.02448.pdf.

[6] Richard Combes, Stefan Magureanu, Alexandre Proutière,Cyrille Laroche. "Learning to Rank: Regret Lower Bounds and Efficient Algorithms". In: (2016). URL: http://rcombes.supelec.free.fr/pdf/sigmetrics_2015_rank.pdf.

[7] Shi Zong, Hao Ni, Kenny Sung,Nan Rosemary Ke,Zheng Wen,Branislav Kveton. "Cascading Bandits for Large-Scale Recommendation Problems". In: (2016). URL: https://arxiv.org/pdf/1603.05359.pdf.

[8] Sumeet Katariya, Branislav Kveton, Csaba Szepesvári,Zheng Wen. "DCM Bandits: Learning to Rank with Multiple Clicks". In: (2016). URL: https://arxiv.org/abs/1602.03146.

[9] Tor Lattimore, Branislav Kveton, Shuai Li, Csaba Szepesvári. "TopRank: A Practical Algorithm for Online Stochastic Ranking". In: (2019). URL: https://arxiv.org/pdf/1806.02248.pdf.

## Appendix A `CascadeUCB1` python implementation

```
import sys
import os
import numpy as np
import pandas as pd

class CascadeUCB():
    def __init__(self, number_of_rounds, L, K):
        super().__init__()
        self.number_of_rounds = number_of_rounds
        self.L = L
        self.K = K
        self.T = np.zeros((number_of_rounds, L))
        self.U = np.zeros((number_of_rounds, L))
        self.w = np.zeros((number_of_rounds, L))
        self.A = np.zeros((number_of_rounds, K), dtype=np.int32)
        self.C = np.zeros(number_of_rounds)
        self.regrets = np.zeros(number_of_rounds)
        return

    def initialize(self, dataset, weights):
        self.T[0, :] = 1
        for t in range(self.L):
            d = np.random.permutation(self.L)
            At = np.append([t], d[d != t][:self.K-1])
            reward = dataset[t][At]
            self.w[0, t] = reward[0]
        # Best reward
        r = 1
        for k in range(self.K):
            r = r*(1-weights[k])
        self.best_f = 1-r

    def f(self, t):
        # best permutation reward
        r = 1
        for k in range(self.K):
            r = r*(1-self.w[t-1, self.A[t, k]])
        reward = 1-r
        return reward

    def update_ucb_item(self, e, t):
        c = np.sqrt((1.5*np.log(t))/self.T[t-1, e])
        return self.w[t-1, e] + c

    def update_weights(self, t):
        self.T[t] = self.T[t-1]
        self.w[t] = self.w[t-1]
        for k in range(min(self.K, int(self.C[t])+1)):
            if k < int(self.C[t]):
                self.T[t, self.A[t, k]] += 1
                self.w[t, self.A[t, k]] = (
                    self.T[t-1, self.A[t, k]]*self.w[t-1, self.A[t, k]])/self.T[t, self.A[t, k]]
```

```
        else:
            self.T[t, self.A[t, k]] += 1
            self.w[t, self.A[t, k]] = (
                self.T[t-1, self.A[t, k]]*self.w[t-1, self.A[t, k]]+1)/self.T[t, self.A[t, k]]

def one_round(self, t, dataset):
    self.U[t] = [self.update_ucb_item(e, t)
                 for e in range(self.L)]
    self.A[t] = np.argsort(self.U[t])[-self.K:][::-1]
    # get reward
    reward = dataset[t][self.A[t]]
    # compute regret
    immediate_regret = self.best_f-self.f(t)
    self.regrets[t] = self.regrets[t-1] + np.abs(immediate_regret)
    # get index of  attractive item
    if np.sum(reward) > 0:
        self.C[t] = np.argmax(reward == 1)
    else:
        self.C[t] = 1e6
    self.update_weights(t)
    return True
```