# LinUBC online implementation

Abhishek Bhatia

September 10, 2018

## 1 Problem statement and Motivation

The problem of personalized news article recommendation can be modeled as a multi-armed bandit problem with context information. Formally, a contextual-bandit algorithm $A$ proceeds in discrete trials t = 1, 2, 3, . . . In trial $t$:

1. The LinUSB algorithm observes the current user $u_t$ and a set $A_t$ of arms or actions together with their feature vectors $x_{t,a}$ for $a \in A_t$. This vector $x_{t,a}$ summarizes information of both $u_t$ and arm $a$, and will be referred to as the context.

2. Based on observed payoffs in previous trails, the algorithm $A$ chooses an arm $a_t \in A_t$, and consequently receives payoff $r_{t,a}$ whose expectation depends on both user $u_t$ and the arm $a_t$.

3. The algorithm then improves its arm-selection technique with the new observation obtained, that is, $(x_{t,a}, a_t, r_{t,a_t})$.

When this problem is viewed in the context of article recommendation, articles are the arms. When the article that was presented is clicked, a reward of 1 is obtained; otherwise, the payoff is 0. The expected reward of an article is then ts clickthrough rate (CTR), and by choosing an article with maximum CTR is the same as maximizing the total expected reward in previous mentioned bandit formulation.

## 2 Dataset

- The personalization dataset consists of t =1,. . . , 10000 data points.

- The dataset contains the action($a_t$) was performed at each timestep(t). The action $a_t \in \{1,\ldots,10\}$.
- For each time-step, the dataset contains the reward that was obtained $y_t \in \{0,1\}$.
- There are 100 context features which is represented as a vector $x_t \in R^{100}$.

# 3 Algorithm

In Algorithm 1, we give the algorithm used. Here, we use a LinUCB with disjoint linear models which is modified to work in an online manner on the dataset given. We then step through the stream of logged events in the dataset one by one. If, given the current history $h_{t-1}$, it happens that the policy selects the same arm $a$ as the one that was selected by the logging policy, then the corresponding event is retained. Otherwise, if the policy selects a different arm from the one that was taken by the logging policy, then the event is entirely ignored, and the algorithm proceeds to the next event without any other change in its state. Also, the context is same in our case for all actions.

**Data:** Inputs: $\alpha \in$ R+,dataset

**for** $t$ = *1,2,3 ....T* **do**

  Observe features of all arms a $\in A_t$: $x_t$, $a \in R^d$;

  **for** *a in* $A_t$ **do**

    **if** *a is new* **then**

      $A_a \rightarrow I_d$ (d-dimensional identity matrix);

      $b_a \rightarrow 0_{d\times 1}$ (d-dimensional zero vector);

    **end**

    $\theta \rightarrow A_a^{-1} b_a$;

    $p_{t,a} \rightarrow \theta^T x_{t,a} + \alpha\sqrt{x_{t,a}^T A_a^{-1} x_{t,a}}$;

  **end**

  Choose arm $a_t = \arg\max_{a\in A_t} p_{t,a}$ with ties broken arbitrarily;

  compute C(t) ;

  **if** $a_t$*is* $a_t^{dataset}$ **then**

    $A_a \rightarrow A_{a_t} + x_{t,a_t} x_{t,a_t}^T$;

    $b_a \rightarrow b_{a_t} + r_t x_{t,a_t}$;

  **end**

**end**

**Algorithm 1:** linUSB

# 4 Metric for evaluation

In this report, we use cumulative take-rate of our actions for any time t as evaluation metric for the data-set. This is computed as go through the dataset.

**Notation**

$\pi_{t-1}$ : algorithm trained on data upto time t-1.

$\pi_{t-1}(x_t)$ : action that algorithm chooses for the context $x_t$ which it observes at time $t$.

$a_t$ : real action that taken in the data-set at time t.

$y_t$ : real reward that was obtained at time t.

$$C(T) = \frac{\sum_{t=1}^{T} y_t \times \mathbf{1}[\pi_{t-1}(x_t) = a_t]}{\sum_{t=1}^{T} \mathbf{1}[\pi_{t-1}(x_t) = a_t]}$$

# 5 Experimental Setup and Optimizations

- We load the dataset and perform linUSB as given in 1 for a particular alpha strategy.
- Then we plot C(t) for each time-step for each alpha strategy on a graph to compare.

We used the following alpha strategies:

- $\alpha(t) = \frac{1}{t}$
- $\alpha(t) = \frac{1}{\sqrt{t}}$
- $\alpha(t) = \frac{0.1}{\sqrt{t}}$
- $\alpha(t) = 0.1$
- $\alpha(t) = e^{-t}$

The strategy number 3 listed above was determined by doing a grid-search on the numerator value.

# 6 Results

## 6.1 Cumulative take-rate replay for different alpha strategies

In Table 1, we show the Cumulative take-rate replay(C(T)) at last time step for different alpha strategies. We see that, the best C(T) is 0.9458.
In the Figure 1, we plot C(T) as time-steps progress for different alpha strategies. We notice $\alpha = \frac{0.1}{\sqrt{t}}$ achieves the best C(T), followed by $\alpha = \frac{1}{\sqrt{t}}$ and $\alpha = \frac{1}{t}$. In comparsion to these strategies, $\alpha = e^{t}$ and $\alpha = 0.1$ converge relatively slowly and achieve sub-optimal C(T)s throughout and finally.
In addition, we can see that both $\alpha=\frac{1}{\sqrt{t}}$ and $\alpha=\frac{1}{t}$ achieve the same final C(T) although $\alpha=\frac{1}{t}$ is noted to converge much faster.

# 7 Code

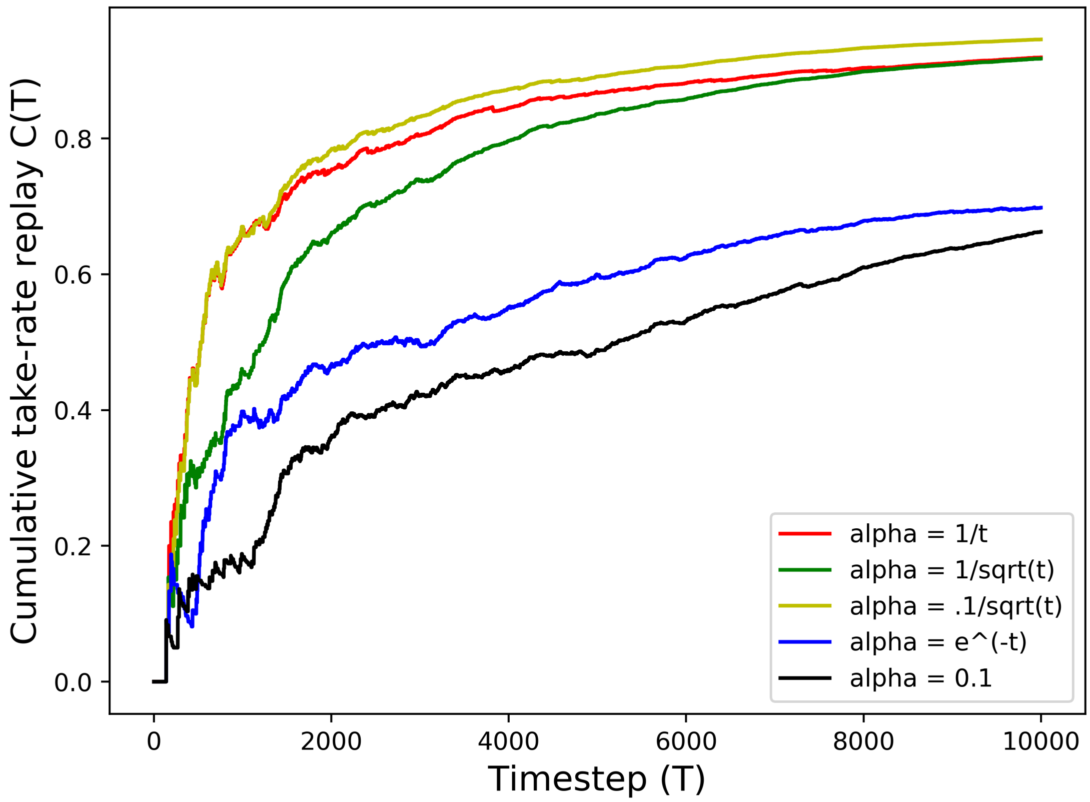


Figure 1: Cumulative take-rate replay with timesteps for different strategies of alpha

| alpha | C(T) |
|---|---|
| $\frac{1}{t}$ | 0.919417 |
| $\frac{1}{\sqrt{t}}$ | 0.917636 |
| $\frac{0.1}{\sqrt{t}}$ | **0.945894** |
| $e^{-t}$ | 0.698150 |
| 0.1 | 0.662687 |

Table 1: Final Cumulative take-rate replay for different strategies of alpha

```python
import pandas as pd
import numpy as np
from numpy.linalg import inv
from math import sqrt
import ipdb
from tqdm import tqdm
import matplotlib.pyplot as plt
import math

def read_data(filename):
    '''
    Input: path to csv file
    Output: numpy arrays of action, reward and context
    Description: Read csv to dataframe. Parses the dataframe
                 and converts to numpy arrays.
    '''
    df = pd.read_csv(filename, header=None, sep="\s+")

    action = df.iloc[:,0].values
    reward = df.iloc[:,1].values
    x_ta = df.iloc[:,2:].values

    return action, reward, x_ta


def initialize_params(d,num_actions):
    '''
    Input: time-steps and number of dimensions
    Output: A_a and b_a
    Description: Initialize A_a to identity matrix for each action.
                 Initialize b_a to 0 d-dimensional vector
    '''
    A = [np.identity(d) for i in range(num_actions)]
    b_as = [np.zeros(d) for i in range(num_actions)]

    return A , b_as

def linUCB(action, reward, x_ta, A, b_as, alphas):
    ''''
    Input:  action, reward and context features for each timestep in the dataset.
            Initialized values for A and b_as.
            Alphas are each timestep for a particular alpha strategy.
    Output: C(T) as an array of size Tx1, where T is the number of
            time-steps in the dataset.
    Description: perform a LinUCB with disjoint linear models which is
                 modified to work in an online manner on the dataset given.
                 And compute C(t) at each timestep t.
    '''

    c_t_num = []
```

```
    c_t_den = []

    for i in tqdm(range(time_steps)):
        alpha = alphas[i]
        x_ta_i = x_ta[i,:]
        r_i = reward[i]
        action_i = action[i]

        p_t_a=[0.]*10
        for a_i in range(10):
            inv_a = inv(A[a_i])
            theta = np.dot(inv_a,b_as[a_i])
            p_t_a[a_i] = np.dot(theta.T,x_ta_i) + alpha * \
                        sqrt(np.dot(np.dot(x_ta_i.T,inv_a),x_ta_i))

        # choose the best action
        a_t = np.argmax(p_t_a)

        if i>=1:
            # update
            if a_t+1 == action_i:

                x_ta_i= x_ta_i.reshape(100,1)
                A[a_t] +=  np.dot(x_ta_i,x_ta_i.T)
                b_as[a_t] +=  r_i*x_ta_i.flatten()

                c_t_num.append(r_i)
                c_t_den.append(1.)
            else:
                c_t_num.append(0.)
                c_t_den.append(0.)

    c_t_num = np.cumsum(c_t_num)
    c_t_den = np.cumsum(c_t_den)

    np.seterr(all='ignore')
    c_t = np.nan_to_num(np.divide(c_t_num,c_t_den, dtype=float))

    return c_t

def plot_ct(c_t_1_t, c_t_sqrt,c_01sqrt, c_e_t, c_point01):
    '''
    Input: C(T) for alpha strategies
    Description: Plot the C(T) for each alpha strategy as time-steps progress.
                 The resulting plot is saved on disk
    '''
    p1, = plt.plot(range(1,len(c_t_1_t)+1),c_t_1_t,'r',
        label="alpha = 1/t")
    p2, = plt.plot(range(1,len(c_t_sqrt)+1),c_t_sqrt,'g',
        label="alpha = 1/sqrt(t)")
```

```
    p3, = plt.plot(range(1,len(c_01sqrt)+1),c_01sqrt,'y',
        label="alpha = .1/sqrt(t)")
    p4, = plt.plot(range(1,len(c_e_t)+1),c_e_t,'b',
        label="alpha = e^(-t)")
    p5, = plt.plot(range(1,len(c_point01)+1),c_point01,'k',
        label="alpha = 0.1")

    plt.legend(handles=[p1, p2, p3, p4, p5])


    plt.xlabel("Timestep (T)",fontsize=14)
    plt.ylabel("Cumulative take-rate replay C(T)",fontsize=14)
    plt.tight_layout()
    plt.savefig("plots", dpi=500)
    plt.close()

# read the dataset
action, reward, x_ta = read_data("dataset.txt")

# determine dataset specific values
d = x_ta.shape[1]
time_steps = action.shape[0]
num_actions = 10

# use different alpha strategies
alphas_sqrt = [1./sqrt(i+1.0) for i in range(time_steps)]
alphas_01sqrt = [0.1/sqrt(i+1.0) for i in range(time_steps)]
alphas_1_t = [1./(i+1.0) for i in range(time_steps)]
alphas_e_t = [math.exp(-i) for i in range(time_steps)]
alphas_point01 = [0.1 for i in range(time_steps)]

# run linUCB for different alphas
A_init , b_as_init = initialize_params(d,num_actions)
c_t_1_t = linUCB(action, reward, x_ta, A_init , b_as_init, alphas_1_t)
print("Final C(T) %s %f " %("alpha = 1/t", c_t_1_t[-1]))

A_init , b_as_init = initialize_params(d,num_actions)
c_t_sqrt = linUCB(action, reward, x_ta, A_init , b_as_init, alphas_sqrt)
print("Final C(T) %s %f " %("alpha = 1/sqrt(t)", c_t_sqrt[-1]))

A_init , b_as_init = initialize_params(d,num_actions)
c_e_t = linUCB(action, reward, x_ta, A_init , b_as_init, alphas_e_t)
print("Final C(T) %s %f " %("alpha = e^(-t)", c_e_t[-1]))


A_init , b_as_init = initialize_params(d,num_actions)
c_point01 = linUCB(action, reward, x_ta, A_init , b_as_init, alphas_point01)
print("Final C(T) %s %f " %("alpha = 0.1", c_point01[-1]))
```

```
A_init , b_as_init = initialize_params(d,num_actions)
c_01sqrt = linUCB(action, reward, x_ta, A_init , b_as_init, alphas_01sqrt)
print("Final C(T) %s %f " %("alpha = .1/sqrt(t)", c_01sqrt[-1]))

# plot the result for C(T)
plot_ct(c_t_1_t, c_t_sqrt,c_01sqrt, c_e_t, c_point01)
```