Project 4
ESE 545, Data Mining: Learning from Massive Datasets
Team members: Shuairui Yao, Penn ID: 47673675; Zhefu Peng, Penn ID: 29419150

Introduction

The purpose of this project is to learn to process a massive dataset with programming techniques in practice, using Python to implement algorithms, about recommender system, such as *ε-greedy* , UCB, Thompson Sampling, with their improved version, which could help us develop better clusters, getting better performance.

Problem1 Data Importing and Problem Formulating

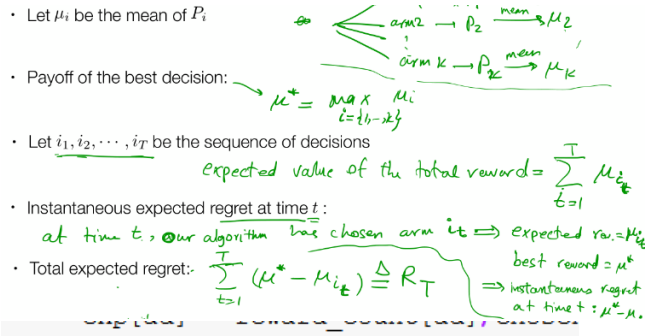
As the problem mentions, we need to import data, and we need to formulate the problem by ourselves in order to decide which algorithm we should implement. As the problem mentions, each row represents a kind of advertisement, and each column represents a round, and the value would imply whether the user click the ad at this round. Here are what we do:

i. Load dataset, using pandas to load csv file, and we can see the dimension of dataset is 50 * 32657. That means the dataset contains 50 kinds of advertisement and try it for 32657 rounds.

```
In [1]: # read data
import pandas as pd
import random
import numpy as np
import time
data = pd.read_csv('yahoo_ad_clicks.csv',header = None)
kt = data.shape # 50 ads; 32657 rounds
k = kt[0]
T = kt[1]
print("k",k)
print("T",T)
data = np.array(data)
```

```
k 50
T 32657
```

ii. Then, we need to process the data. From the notes, we know that when we do recommender system of advertisement, the reward should be [0,1], that means 0 or 1. Viewing the dataset, indication of reward is already 0 or 1. However, since we need to evaluate each algorithm's performance, we need to calculate the best μ in the dataset. And the best μ is changed in each round.




```
## Calculating regret using miu
best_miu = exp[np.argmax(exp)]
```

The parameter "best_miu" would help us to calculate the regret after each round.

Problem2 Design and Choose a Strategy with Partial Feedback to Solve Problem.
As the problem mentions, we need to try and test all algorithm we learned from the class, to find the fittest one.

i. First one we try is *ε-greedy* , using the parameter best_miu to calculate the regret. We designed it based on the note:

- For $t = 1 : T$
  - Set $\epsilon_t = O(1/t)$
  - With probability ϵ_t : Explore by deciding uniformly at random
  - With probability $1 - \epsilon_t$: Exploit by picking decision with highest empirical mean payoff

First, we try to set the propriate value of ε_t , since t is from 1 to 32657, changed a lot during the solving, we decide to use log function to calculate ε_t .

```
    e1 = 1/math.log2(t+5)
```

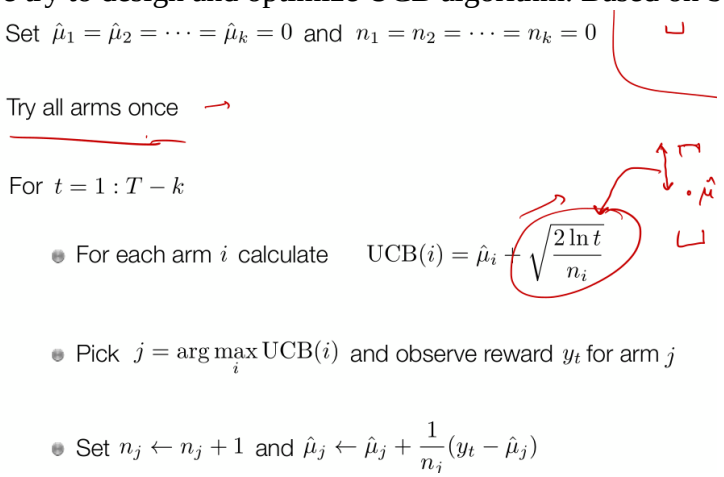
And, in order to converge at a better result, we decide to optimize the algorithm, that in the first certain number of rounds, try to explore more, but exploit less. To get a better pay-off.

```
for t in range(0,T):
    ## In the first range, explore more
    e = 1/np.log(t+5)
    p = np.random.uniform(0,1)
    if p <= e:
        idx = np.random.randint(0,50) ## explore
    else:
        idx = np.argmax(exp) ## exploit
```

Above is what we optimized for the *ε-greedy* algorithm, here is the code for it:

```
## e-greedy
import math
def e_greedy(data):
    kt = data.shape
    k = kt[0]
    T = kt[1]
    chosen_count = np.zeros(k)
    reward_count = np.zeros(k)
    exp = np.zeros(k)
    actual_mu = []
    for t in range(0,T):
        ## In the first range, explore more
        e = 1/np.log(t+5)
        p = np.random.uniform(0,1)
        if p <= e:
            idx = np.random.randint(0,50) ## explore
        else:
            idx = np.argmax(exp) ## exploit
        reward = data[idx,t]
        chosen_count[idx] += 1
        if reward == 1:
            reward_count[idx] += 1
        exp[idx] = reward_count[idx]/chosen_count[idx]
        actual_mu.append(exp[idx])
    # calculate regret
    best_miu = exp[np.argmax(exp)]
    diff_miu = 0
    regret = []
    for t in range(0,T):
        diff_miu += (best_miu-actual_mu[t])
        regret.append(diff_miu/(t+1))
    return regret, np.sum(reward_count,axis = 0)
```

ii. Now we try to design and optimize UCB algorithm. Based on below:



The point is, the number of rounds in UCB is T-k, so we need to use the last k columns, to help us try all arms once. Thus, we decide to find the best μ in the last k rounds, as the estimated μ we know. And using estimated μ, to decide after trying all arms (ads) once, the reward for each ad is 1 or 0, with probability we calculated before.

```
def try_all_ad_once(data):
    kt = data.shape
    k = kt[0]
    T = kt[1]
    ## Getting the initial estimated_miu
    estimated_miu = np.zeros(k)
    prob = np.sum(data[:,T-k:],axis=1)/k
    for i in range(k):
        ## The probability that this row becomes 1
        estimated_miu[i] = np.random.binomial(n=1, p=prob[i])
    return estimated_miu
```

Then, we need to define a function to calculate the bonus in each round.

```
## define a function to calculate bonus in each round
def calculate_bonus(r, ads): ## r means the round, ads means the advertisement
    if chosen_count[ads] == 0:
        return 1
    else:
        return np.sqrt(2 * np.log(r) / chosen_count[ads])
```

Here is the all code for UCB algorithm:

```
## Starting UCB algorithm
def UCB1(data):
    kt = data.shape
    k = kt[0]
    T = kt[1]
    chosen_count = np.ones(k)
    reward_count = np.zeros(k)
    exp = np.zeros(k)
    total_reward = 0
    best_loss = 0
    actual_loss = 0
    estimated_miu = try_all_ad_once(data)
    actual_miu = []
    for t in range(1,(T-k+1)):
        ## Calculate each ad's value of ucb
        upper_bound_probs = [estimated_miu[item] + calculate_bonus(t, item, chosen_count) for item in range(k)]
        ## Pick the ad who has max value
        ad = np.argmax(upper_bound_probs)
        ## Observe the rewards (not sure), reward is 0 or 1.
        reward = data[:,t][ad]
        ## Updating the counts of the ad selected
        chosen_count[ad] += 1
        ## Updating the estimated value the ad selected
        estimated_miu[ad] = ((chosen_count[ad]-1)*estimated_miu[ad]+reward)/chosen_count[ad]
        actual_miu.append(estimated_miu[ad])
        ## Recording the number of rewards
        total_reward += reward
        if reward == 1:
            reward_count[ad] += 1
        exp[ad] = reward_count[ad]/chosen_count[ad]
    ## Calculating regret using miu
    best_miu = exp[np.argmax(exp)]
    dif_miu = 0
    regret = []
    for t in range(1,T-k+1):
        dif_miu += best_miu - actual_miu[t-1]
        regret.append(dif_miu/t)
    return regret,total_reward
```

iii. Next, we try to design and optimize Thompson algorithm. Since Thompson Sampling is most used, in practice, we hope Thompson could perform best. And we write down the code based on below:

Algorithm 1 Thompson Sampling for Bernoulli bandits

For each arm $i = 1, \ldots, N$ set $S_i = 0, F_i = 0$.

foreach $t = 1, 2, \ldots,$ **do**

  For each arm $i = 1, \ldots, N$, sample $\theta_i(t)$ from the Beta$(S_i + 1, F_i + 1)$ distribution.

  Play arm $i(t) := \arg\max_i \theta_i(t)$ and observe reward r_t.

  If $r = 1$, then $S_{i(t)} = S_{i(t)} + 1$, else $F_{i(t)} = F_{i(t)} + 1$.

end

And, here is the code, the value of s(i) and f(i) affects explore more or exploit more in this round.

```
# Thompson
def thompson(data):
    kt = data.shape
    k = kt[0]
    T = kt[1]
    S = [0]*k
    F = [0]*k
    beta = []
    reward = 0
    chosen_count = np.ones(k)
    reward_count = np.zeros(k)
    best_arm = np.argmax(np.sum(data,axis=1))
    best_loss = 0
    actual_loss = 0
    regret = []
    actual_miu = []
    for t in range(0,T):
        # select ad
        theta_t = []
        for i in range(0,k):
            theta_t.append(np.random.beta(S[i]+1,F[i]+1))
        chosen_arm = np.argmax(theta_t)

        this_reward = data[chosen_arm,t]
        reward += this_reward
        if(this_reward == 1):
            S[chosen_arm] += 1
        else:
            F[chosen_arm] += 1
        actual_miu.append(S[chosen_arm]/(S[chosen_arm] + F[chosen_arm]))
    dif_miu = 0
    best_idx = np.argmax(np.array(S)/(np.array(S)+np.array(F)))
    best_miu = np.array(S[best_idx])/(np.array(S[best_idx])+np.array(F[best_idx]))
    for t in range(0,T):
        dif_miu += best_miu - actual_miu[t]
        regret.append(dif_miu/(t+1))
    return regret,reward
```

iv. The last algorithm we decide to try is exp3, which is a kind of non-stochastic algorithm, with limited feedback. We based on:

Exp3 (Exponential weights for Exploration and Exploitation)
Parameter: a nonincreasing sequence of real numbers $(\eta_t)_{t\in\mathbb{N}}$.
Let p_1 be the uniform distribution over $\{1,\dots,K\}$.
For each round $t=1,2,\dots,n$

(1) Draw an arm I_t from the probability distribution p_t.
(2) For each arm $i=1,\dots,K$ compute the estimated loss $\widetilde{\ell}_{i,t}=\frac{\ell_{i,t}}{p_{i,t}}\mathbb{1}_{I_t=i}$ and update the estimated cumulative loss $\widetilde{L}_{i,t}=\widetilde{L}_{i,t-1}+\widetilde{\ell}_{i,s}$.
(3) Compute the new probability distribution over arms $p_{t+1}=(p_{1,t+1},\dots,p_{K,t+1})$, where

$$p_{i,t+1}=\frac{\exp(-\eta_t\widetilde{L}_{i,t})}{\sum_{k=1}^{K}\exp(-\eta_t\widetilde{L}_{k,t})}.$$

What's more, different from the above three algorithms, for Exp3, since it's a non-stochastic algorithm, the way to calculate the regret need to be changed, using loss:

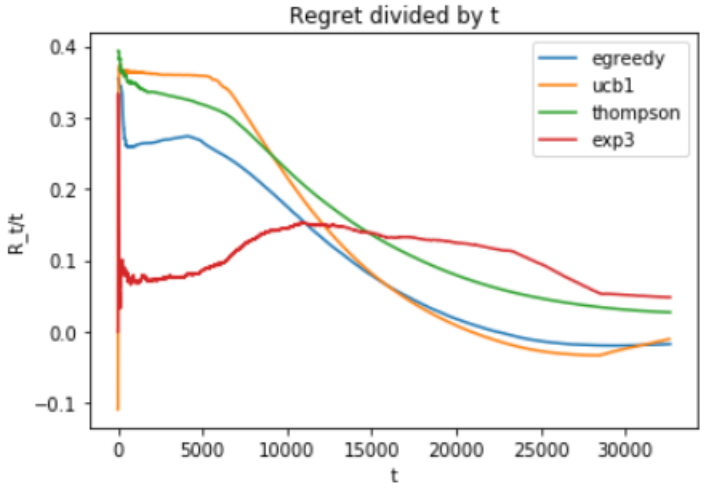
```
for t in range(0,T):
    chosen_arm = choices(range(0,k),p_t)[0]
    cumu_L += 1 - data[:,t]
    best_arm = np.argmin(cumu_L)
    this_reward = data[chosen_arm,t]
    reward += this_reward
    actual_loss += 1 - data[chosen_arm,t]
    best_loss = cumu_L[best_arm]

    dif_loss = actual_loss - best_loss
    regret.append(dif_loss/(t+1))
```

And here is the code:

```
from random import choices
def exp3(data):
    kt = data.shape # 50 ads; 32657 rounds
    k = kt[0]
    T = kt[1]
    ita_t = list(map(lambda x: np.sqrt(np.log(k)/(k*x)), range(1,T+1)))
    p_t = [1/k]*k
    L = [0]*k
    cumu_L = [0]*k
    reward = 0
    regret = []
    dif_loss = 0
    best_loss = 0
    actual_loss = 0
    for t in range(0,T):
        chosen_arm = choices(range(0,k),p_t)[0]
        cumu_L += 1 - data[:,t]
        best_arm = np.argmin(cumu_L)
        this_reward = data[chosen_arm,t]
        reward += this_reward
        actual_loss += 1 - data[chosen_arm,t]
        best_loss = cumu_L[best_arm]

        dif_loss = actual_loss - best_loss
        regret.append(dif_loss/(t+1))

        l_it_hat = (1 - data[chosen_arm,t])/p_t[chosen_arm]
        L[chosen_arm] += l_it_hat
        # update p_t
        vec = np.exp(-ita_t[t]*np.array(L))
        p_t = vec/sum(vec)
    return regret,reward
```

v. Now that we have designed and tried 4 kinds of algorithm, such as *ε-greedy* , UCB, Thompson Sampling and EXP3. We now need to plot the regret of each kind, to see whose performance is the best.

```
from matplotlib import pyplot as plt
plt.plot(range(1,T+1),regret_egreedy)
plt.plot(range(1,T-k+1),regret_ucb1)
plt.plot(range(1,T+1),regret_thompson)
plt.plot(range(1,T+1),regret_exp3)
plt.legend(labels = ['egreedy','ucb1','thompson','exp3'], loc = 'best')
plt.xlabel('t')
plt.ylabel('R_t/t')
plt.title("Regret divided by t")
plt.show()
```



We can see each algorithm's performance, and we would talk about it later in problem4.

Problem3 Design a strategy to solve your multi-armed bandit problem with full feedback

The purpose of this problem is to design or optimize an algorithm, to solve the problem.

i. We decide to implement Multiplicative weight update, which is designed for full feedback, and based on below:

Multiplicative Weights algorithm

Initialization: Fix an $\eta \leq \frac{1}{2}$. With each decision i, associate the weight $w_i^{(1)} := 1$.
For $t = 1, 2, \ldots, T$:

1. Choose decision i with probability proportional to its weight $w_i^{(t)}$. I. e., use the distribution over decisions $\mathbf{p}^{(t)} = \{w_1^{(t)}/\Phi^{(t)}, \ldots, w_n^{(t)}/\Phi^{(t)}\}$ where $\Phi^{(t)} = \sum_i w_i^{(t)}$.
2. Observe the costs of the decisions $\mathbf{L}^{(t)}$
3. Penalize the costly decisions by updating their weights as follows: for every decision i, set

$$w_i^{(t+1)} = w_i^{(t)}(1 - \eta \ell_{i,t})$$

The same as exp3 is, both MWA and exp3 is non-stochastic, they need to use loss to calculate regret:

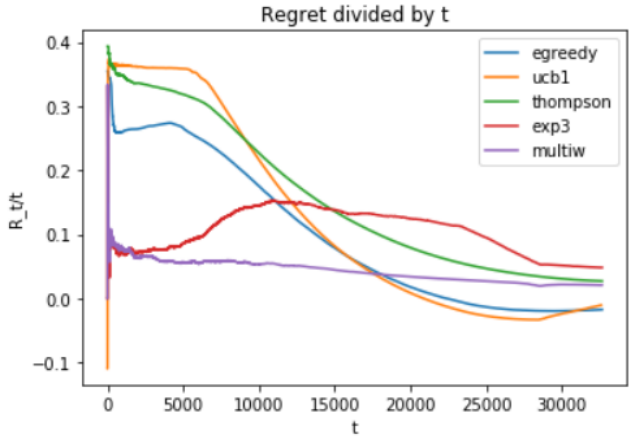
```
actual_loss += 1 - data[chosen_arm,t]
best_loss = cumu_L[best_arm]

dif_loss = actual_loss - best_loss
regret.append(dif_loss/(t+1))
```

And, here is the code:

```
# multiplicative weight updates (full feedback)
def multiWeight(data):
    kt = data.shape # 50 ads; 32657 rounds
    k = kt[0]
    T = kt[1]
    ita_t = 1/np.sqrt(T)
    p_t = [0]*k
    w = [1]*k
    cumu_L = [0]*k
    reward = 0
    regret = []
    dif_loss = 0
    best_loss = 0
    actual_loss = 0
    for t in range(0,T):
        p_t = np.divide(w,sum(w))
        chosen_arm = choices(range(0,k),p_t)[0]
        cumu_L += 1 - data[:,t]
        best_arm = np.argmin(cumu_L)
        this_reward = data[chosen_arm,t]
        reward += this_reward

        actual_loss += 1 - data[chosen_arm,t]
        best_loss = cumu_L[best_arm]

        dif_loss = actual_loss - best_loss
        regret.append(dif_loss/(t+1))

        # observe the costs of decisions L at that round
        L = 1 - data[:,t]
        # penalize the costly decisions by updating their weights as follows
        w = np.multiply(w,(1-ita_t*np.array(L)))
    return regret, reward
```

ii. Now we need to compare the result and performance with the algorithm we tried in problem2.

```
In [20]: # regret
from matplotlib import pyplot as plt
plt.plot(range(1,T+1),regret_egreedy)
plt.plot(range(1,T-k+1),regret_ucb1)
plt.plot(range(1,T+1),regret_thompson)
plt.plot(range(1,T+1),regret_exp3)
plt.plot(range(1,T+1),regret_multiw)
plt.legend(labels = ['egreedy','ucb1','thompson','exp3','multiw'], loc = 'best')
plt.xlabel('t')
plt.ylabel('R_t/t')
plt.title("Regret divided by t")
plt.show()
```

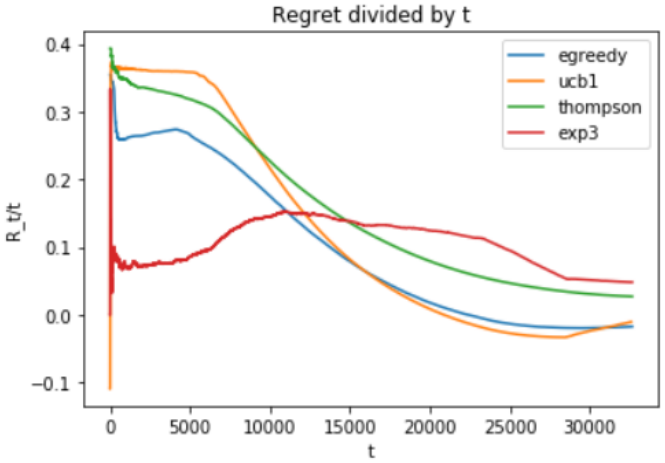


iii. From the picture, we can find that after T= 32657 rounds, the multiplicate weighted update algorithm gets the best performance, and converge faster. I think reasons for the best performance of MWA maybe its full back, he could find the best option in every-time choose. What's more, using the MWA algorithm, we have the limited number of loss, and decrease the cost as much as possible.

Problem4 Explanation about Algorithm We choose

i. First, we already got the result of comparison.

```
from matplotlib import pyplot as plt
plt.plot(range(1,T+1),regret_egreedy)
plt.plot(range(1,T-k+1),regret_ucb1)
plt.plot(range(1,T+1),regret_thompson)
plt.plot(range(1,T+1),regret_exp3)
plt.legend(labels = ['egreedy','ucb1','thompson','exp3'], loc = 'best')
plt.xlabel('t')
plt.ylabel('R_t/t')
plt.title("Regret divided by t")
plt.show()
```



ii. From the plot, we choose **Thompson Sampling**, since its performance of regret is the best (tend to zero, and not be negative). What's more, though its convergence speed is not the best, but also faster than other 2 algorithms (e-greedy and exp3).

For the reasons of Thompson Sampling perform best:

1. In the beginning of process, each arm's sum of (s(i)+f(i)) is small, and the beta distribution is wide, the random number would be greater, so the probability of exploration would be increased;
2. When the number of rounds becomes more, each arm's sum of (s(i)+f(i)) is large, and the beta distribution is narrow, the profit of each arm is confirmed;
3. If for one arm, sum of (s(i)+f(i)) is large, s(i) / (s(i)+f(i)) is large, so the distribution for this arm is narrow, and near to center situation, in each round, it would have priority in each round;
4. For UCB algorithm, since it needs to try all arms once, it will cost time, and it's still a "confirmed" algorithm, with limited exploration ability.