Reinforcement Learning: Homework #2

Due on April 7, 2020 at 11:59pm

Professor Ziyu Shao

Yiwei Yang
2018533218

Problem 1

1. First denote $Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\Sigma_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$,where a is the action.
2. Then denote $\alpha > 0$ is a step-size parameter
3. Also, $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up through and including time t, which can be computed incrementally.

Algorithm 1: Gradient Bandit in 3-armed Bernoulli bandit problem

Initialize: $H_1(a) = 0, R_t = 0b$ *for baseline*

```
#Sample model
m for t=1,2...,N do
    for j ∈ {1,2,3} do
        Sample A_t from π_t(j)
    I(t) ← A_t
    R_t ← r_I
    Baseline ← b
    for j ∈ {1,2,3} do
        if A_t == j then
            H_{t+1}(j) ← H_t(j) + α(R_t − Baseline)(1 − π_t(j))
        else
            H_{t+1}(j) ← H_t(j) + α(R_t − Baseline)(π_t(j))
```

Problem 2

Basic Settings

I. Environment Imports

Personally, I utilize `numpy` to generate random variables and `matplotlib` to plot the graph.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
import scipy,time,sys
import scipy.stats as stats
from scipy.stats import beta
plt.style.use('seaborn')
np.random.seed(5678)
np.set_printoptions(3)
```

II. Denotions

As declared in the requirements, we use the Oracle values listed below.

$Arm\ j$	0	1	2
$Oracle\ \theta_j$	0.9	0.8	0.7

The maximum θ_j will be 0.9. The theoretically maximized expectation of aggregate rewards over $N = 5000$ time slots is $\theta_1 N = 4500$

[2]:
```
# setting the ground truth
num_bandit = 3
num_ep  = 1000
num_iter= 5000
gt_prob = np.array([0.9,0.8,0.7])
optimal_choice = np.argmax(gt_prob)
print(gt_prob)
print('Best Choice: ',optimal_choice,gt_prob[optimal_choice])
```

```
[0.9 0.8 0.7]
Best Choice:  0 0.9
```

Function `pull(arm)` is defined to give the result of the bandit regarding Oracle values. `1` means success and `0` stands for failure. The index is from 0 to 2. Also, we define the simulation is from 0 to $N - 1$=4999, and for all the algorithms, we have $K = 1000$ times and we get the average of the results.

[3]:
```
def pull(arm):
    return float(random.random() < gt_prob[arm])
```

III. $\epsilon - Greedy$ Algorithm

The idea of the greedy algorithm is very straightforward:

1. Use past data to estimate a model.
2. Choose actions that can optimize the estimated model.

[4]:
```
# a vectorized
a_expect = np.zeros((num_ep,num_bandit))

for eps in range(num_ep):
    temp_expect = np.zeros(num_bandit)
    temp_choice = np.zeros(num_bandit)

    for iter in range(num_iter//10):
        temp_choice    = temp_choice + 1
        current_reward = np.random.uniform(0,1,num_bandit) < gt_prob
        temp_expect    = temp_expect + current_reward

    a_expect[eps,:] = temp_expect/temp_choice
```

```
print('Ground Truth')
print(gt_prob)
print('Expected ')
print(a_expect.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.9   0.803 0.698]
```

[5]:
```
# b e greedy 0.1
b_pull_count   = np.zeros((num_ep,num_bandit))
b_estimation   = np.zeros((num_ep,num_bandit))
b_reward       = np.zeros((num_ep,num_iter))
b_optimal_pull = np.zeros((num_ep,num_iter))
b_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    epsilon = 0.1
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_expect) if epsilon < np.random.uniform(0,1)␣
 ↪else np.random.choice(np.arange(num_bandit))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
 ↪(temp_pull_count[current_choice]+1)) *␣
 ↪(current_reward-temp_estimation[current_choice])

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    b_pull_count[eps,:]   = temp_pull_count
    b_estimation[eps,:]   = temp_estimation
    b_reward[eps,:]       = temp_reward
    b_optimal_pull[eps,:] = temp_optimal_pull
    b_regret_total[eps,:] = temp_regret
```

```
print('Ground Truth')
print(gt_prob)
print('Expected ')
print(b_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.899 0.796 0.692]
```

[6]:
```
# c e greedy 0.5
c_pull_count   = np.zeros((num_ep,num_bandit))
c_estimation   = np.zeros((num_ep,num_bandit))
c_reward       = np.zeros((num_ep,num_iter))
c_optimal_pull = np.zeros((num_ep,num_iter))
c_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    epsilon = 0.5
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_expect) if epsilon < np.random.uniform(0,1)␣
 ↪else np.random.choice(np.arange(num_bandit))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
 ↪(temp_pull_count[current_choice]+1)) *␣
 ↪(current_reward-temp_estimation[current_choice])

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    c_pull_count[eps,:]   = temp_pull_count
    c_estimation[eps,:]   = temp_estimation
    c_reward[eps,:]       = temp_reward
    c_optimal_pull[eps,:] = temp_optimal_pull
```

```
    c_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(c_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.9   0.801 0.7  ]
```

[7]:
```
# d e greedy 0.9
d_pull_count   = np.zeros((num_ep,num_bandit))
d_estimation   = np.zeros((num_ep,num_bandit))
d_reward       = np.zeros((num_ep,num_iter))
d_optimal_pull = np.zeros((num_ep,num_iter))
d_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    epsilon = 0.9
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_expect) if epsilon < np.random.uniform(0,1)␣
↪else np.random.choice(np.arange(num_bandit))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
↪(temp_pull_count[current_choice]+1)) *␣
↪(current_reward-temp_estimation[current_choice])

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    d_pull_count[eps,:]   = temp_pull_count
    d_estimation[eps,:]   = temp_estimation
    d_reward[eps,:]       = temp_reward
```

```
    d_optimal_pull[eps,:] = temp_optimal_pull
    d_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(d_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.901 0.798 0.699]
```

IV. *UCB* Algorithm

1. Initialization: Try it on each arm first.
2. Calculate the score of each arm according to the formula, choose the arm with the highest score as the choice using $A_n \doteq argmax_a \left(Q_n(a) + c\sqrt{\frac{log(n)}{N_n(a)}} \right)$
3. $\bar{x}_j(t) + \sqrt{\frac{2ln\ t}{T_{j,t}}}$ Among them, t is the current number of trials, $\bar{x}_j(t)$ is the average return from this arm to the present, and $T_{j,t}$ is the number of subjects in this arm. The plus sign is essentially the standard deviation of the mean.
4. Observe the selection results, update t and $T_{j,t}

[8]:
```
# e UBC 1
e_pull_count   = np.zeros((num_ep,num_bandit))
e_estimation   = np.zeros((num_ep,num_bandit))
e_reward       = np.zeros((num_ep,num_iter))
e_optimal_pull = np.zeros((num_ep,num_iter))
e_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    c = 1
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_estimation + c*np.sqrt(2*np.log(iter+1)/
 ↪(temp_pull_count+1)))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
 ↪(temp_pull_count[current_choice]+1)) *␣
 ↪(current_reward-temp_estimation[current_choice])
```

```
        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    e_pull_count[eps,:]   = temp_pull_count
    e_estimation[eps,:]   = temp_estimation
    e_reward[eps,:]       = temp_reward
    e_optimal_pull[eps,:] = temp_optimal_pull
    e_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(e_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.901 0.795 0.684]
```

[9]:
```
# f UBC 5
f_pull_count   = np.zeros((num_ep,num_bandit))
f_estimation   = np.zeros((num_ep,num_bandit))
f_reward       = np.zeros((num_ep,num_iter))
f_optimal_pull = np.zeros((num_ep,num_iter))
f_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    c = 5
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_estimation + c*np.sqrt(2*np.log(iter+1)/
 ↪(temp_pull_count+1)))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
 ↪(temp_pull_count[current_choice]+1)) *␣
 ↪(current_reward-temp_estimation[current_choice])
```

```
        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    f_pull_count[eps,:]   = temp_pull_count
    f_estimation[eps,:]   = temp_estimation
    f_reward[eps,:]       = temp_reward
    f_optimal_pull[eps,:] = temp_optimal_pull
    f_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(f_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.9   0.798 0.699]
```

[10]:
```
# g UBC 10
g_pull_count   = np.zeros((num_ep,num_bandit))
g_estimation   = np.zeros((num_ep,num_bandit))
g_reward       = np.zeros((num_ep,num_iter))
g_optimal_pull = np.zeros((num_ep,num_iter))
g_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    g = 10
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(temp_estimation + c*np.sqrt(2*np.log(iter+1)/
 ↪(temp_pull_count+1)))
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
 ↪(temp_pull_count[current_choice]+1)) *␣
 ↪(current_reward-temp_estimation[current_choice])

        # update reward and optimal choice
```

```
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    g_pull_count[eps,:]   = temp_pull_count
    g_estimation[eps,:]   = temp_estimation
    g_reward[eps,:]       = temp_reward
    g_optimal_pull[eps,:] = temp_optimal_pull
    g_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(g_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.899 0.8   0.699]
```

V. Thompson Sampling Algorithm

$\theta_j, j \in \{1,2,3\}$, are unknown parameters over $(0,1)$. From the Bayesian perspective, we assume their priors are Beta distributions with givenparameters (α_j, β_j)

[11]:
```
# h Thompson Sampling (beta) ([1,1,1],[1,1,1])
h_pull_count   = np.zeros((num_ep,num_bandit))
h_estimation   = np.zeros((num_ep,num_bandit))
h_reward       = np.zeros((num_ep,num_iter))
h_optimal_pull = np.zeros((num_ep,num_iter))
h_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):

    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    alpha = [1, 1, 1]
    beta  = [1, 1, 1]
    for iter in range(num_iter):

        theta_samples = [stats.beta(a=alpha[i],b=beta[i]).rvs(size=1) for i in␣
↪range(num_bandit)]

        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(theta_samples)
```

```
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] +␣
↪current_reward

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    h_pull_count[eps,:]   = temp_pull_count
    h_estimation[eps,:]   = theta_samples
    h_reward[eps,:]       = temp_reward
    h_optimal_pull[eps,:] = temp_optimal_pull
    h_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(h_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.5   0.466 0.465]
```

[12]:
```
# i Thompson Sampling (beta) ([601,401,2],[401,601,3])
i_pull_count   = np.zeros((num_ep,num_bandit))
i_estimation   = np.zeros((num_ep,num_bandit))
i_reward       = np.zeros((num_ep,num_iter))
i_optimal_pull = np.zeros((num_ep,num_iter))
i_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):

    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    alpha = [601, 401, 2]
    beta  = [401, 601, 3]
    for iter in range(num_iter):

        theta_samples = [stats.beta(a=alpha[i],b=beta[i]).rvs(size=1) for i in␣
↪range(num_bandit)]
```

```
        # select bandit / get reward /increase count / update estimate
        current_choice = np.argmax(theta_samples)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] +␣
↪current_reward

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    i_pull_count[eps,:]   = temp_pull_count
    i_estimation[eps,:]   = theta_samples
    i_reward[eps,:]       = temp_reward
    i_optimal_pull[eps,:] = temp_optimal_pull
    i_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(i_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.601 0.4   0.443]
```

VI. Gradient Bandit Algorithm

1. First denote $Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\Sigma_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$,where a is the action.
2. Then denote $\alpha > 0$ is a step-size parameter
3. Also, $\bar{R}_t \in \mathbb{R}$ is the average of all the rewards up through and including time t, which can be computed incrementally.

[13]:
```
# j gradient (0,1)
j_pull_count   = np.zeros((num_ep,num_bandit))
j_estimation   = np.zeros((num_ep,num_bandit))
j_reward       = np.zeros((num_ep,num_iter))
j_optimal_pull = np.zeros((num_ep,num_iter))
j_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
```

```
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0
    baseline = 1
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    j_pull_count[eps,:]   = temp_pull_count
    j_estimation[eps,:]   = temp_estimation
    j_reward[eps,:]       = temp_reward
    j_optimal_pull[eps,:] = temp_optimal_pull
    j_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(j_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(j_estimation.mean(0)-j_estimation.mean(0).min())/
 ↪(j_estimation.mean(0).max()-j_estimation.mean(0).min()) + gt_prob.min()
)
j_estimation = (gt_prob.max()-gt_prob.min())*(j_estimation.mean(0)-j_estimation.
 ↪mean(0).min())/(j_estimation.mean(0).max()-j_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 0.33   0.33 0.33]
Expected Normalized
[0.9   0.834 0.71  ]
```

```
/opt/anaconda3/envs/my-rdkit-env/lib/python3.6/site-
packages/ipykernel_launcher.py:49: RuntimeWarning: invalid value encountered in
true_divide
/opt/anaconda3/envs/my-rdkit-env/lib/python3.6/site-
packages/ipykernel_launcher.py:51: RuntimeWarning: invalid value encountered in
true_divide
```

[14]:
```
# k gradient (0.8,1)
k_pull_count   = np.zeros((num_ep,num_bandit))
k_estimation   = np.zeros((num_ep,num_bandit))
k_reward       = np.zeros((num_ep,num_iter))
k_optimal_pull = np.zeros((num_ep,num_iter))
k_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 1
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
```

```
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    k_pull_count[eps,:]   = temp_pull_count
    k_estimation[eps,:]   = temp_estimation
    k_reward[eps,:]       = temp_reward
    k_optimal_pull[eps,:] = temp_optimal_pull
    k_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(k_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(k_estimation.mean(0)-k_estimation.mean(0).min())/
 ↪(k_estimation.mean(0).max()-k_estimation.mean(0).min()) + gt_prob.min()
)
k_estimation = (gt_prob.max()-gt_prob.min())*(k_estimation.mean(0)-k_estimation.
 ↪mean(0).min())/(k_estimation.mean(0).max()-k_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 0.7   0.45 -0.15]
Expected Normalized
[0.9   0.842 0.7  ]
```

[15]:
```
# l gradient (5,1)
l_pull_count   = np.zeros((num_ep,num_bandit))
l_estimation   = np.zeros((num_ep,num_bandit))
l_reward       = np.zeros((num_ep,num_iter))
l_optimal_pull = np.zeros((num_ep,num_iter))
l_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 5
    baseline = 1
```

```
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    l_pull_count[eps,:]   = temp_pull_count
    l_estimation[eps,:]   = temp_estimation
    l_reward[eps,:]       = temp_reward
    l_optimal_pull[eps,:] = temp_optimal_pull
    l_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(l_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(l_estimation.mean(0)-l_estimation.mean(0).min())/
 ↪(l_estimation.mean(0).max()-l_estimation.mean(0).min()) + gt_prob.min()
)
l_estimation = (gt_prob.max()-gt_prob.min())*(l_estimation.mean(0)-l_estimation.
 ↪mean(0).min())/(l_estimation.mean(0).max()-l_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 0.04  1.53 -0.57]
```

```
Expected Normalized
[0.758 0.9   0.7  ]
```

[16]:
```
# m gradient (20,1)
m_pull_count   = np.zeros((num_ep,num_bandit))
m_estimation   = np.zeros((num_ep,num_bandit))
m_reward       = np.zeros((num_ep,num_iter))
m_optimal_pull = np.zeros((num_ep,num_iter))
m_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 20
    baseline = 1
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    m_pull_count[eps,:]   = temp_pull_count
    m_estimation[eps,:]   = temp_estimation
    m_reward[eps,:]       = temp_reward
    m_optimal_pull[eps,:] = temp_optimal_pull
```

Problem 2 continued on next page...

```
    m_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(m_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(m_estimation.mean(0)-m_estimation.mean(0).min())/
 ↪(m_estimation.mean(0).max()-m_estimation.mean(0).min()) + gt_prob.min()
)
m_estimation = (gt_prob.max()-gt_prob.min())*(m_estimation.mean(0)-m_estimation.
 ↪mean(0).min())/(m_estimation.mean(0).max()-m_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[-0.13 -0.53  1.67]
Expected Normalized
[0.736 0.7   0.9  ]
```

[46]:
```
# n gradient (0.8,0.2)
n_pull_count   = np.zeros((num_ep,num_bandit))
n_estimation   = np.zeros((num_ep,num_bandit))
n_reward       = np.zeros((num_ep,num_iter))
n_optimal_pull = np.zeros((num_ep,num_iter))
n_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 0.2
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
```

```
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                        (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    n_pull_count[eps,:]   = temp_pull_count
    n_estimation[eps,:]   = temp_estimation
    n_reward[eps,:]       = temp_reward
    n_optimal_pull[eps,:] = temp_optimal_pull
    n_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(n_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(n_estimation.mean(0)-n_estimation.mean(0).min())/
↪(n_estimation.mean(0).max()-n_estimation.mean(0).min()) + gt_prob.min()
)
n_estimation = (gt_prob.max()-gt_prob.min())*(n_estimation.mean(0)-n_estimation.
↪mean(0).min())/(n_estimation.mean(0).max()-n_estimation.mean(0).min()) + gt_prob.
↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 1.37 -0.54  0.17]
Expected Normalized
[0.9   0.7   0.774]
```

[18]:
```
# o gradient (0.8,1)
o_pull_count   = np.zeros((num_ep,num_bandit))
o_estimation   = np.zeros((num_ep,num_bandit))
o_reward       = np.zeros((num_ep,num_iter))
o_optimal_pull = np.zeros((num_ep,num_iter))
o_regret_total = np.zeros((num_ep,num_iter))
```

```
for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 1
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    o_pull_count[eps,:]   = temp_pull_count
    o_estimation[eps,:]   = temp_estimation
    o_reward[eps,:]       = temp_reward
    o_optimal_pull[eps,:] = temp_optimal_pull
    o_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(o_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(o_estimation.mean(0)-o_estimation.mean(0).min())/
↪(o_estimation.mean(0).max()-o_estimation.mean(0).min()) + gt_prob.min()
```

```
)
o_estimation = (gt_prob.max()-gt_prob.min())*(o_estimation.mean(0)-o_estimation.
 ↪mean(0).min())/(o_estimation.mean(0).max()-o_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 1.52  0.24 -0.76]
Expected Normalized
[0.9   0.787 0.7  ]
```

[19]:
```
# p gradient (0.8,2)
p_pull_count   = np.zeros((num_ep,num_bandit))
p_estimation   = np.zeros((num_ep,num_bandit))
p_reward       = np.zeros((num_ep,num_iter))
p_optimal_pull = np.zeros((num_ep,num_iter))
p_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 2
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
```

```
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    p_pull_count[eps,:]   = temp_pull_count
    p_estimation[eps,:]   = temp_estimation
    p_reward[eps,:]       = temp_reward
    p_optimal_pull[eps,:] = temp_optimal_pull
    p_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(p_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(p_estimation.mean(0)-p_estimation.mean(0).min())/
 ↪(p_estimation.mean(0).max()-p_estimation.mean(0).min()) + gt_prob.min()
)
p_estimation = (gt_prob.max()-gt_prob.min())*(p_estimation.mean(0)-p_estimation.
 ↪mean(0).min())/(p_estimation.mean(0).max()-p_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.07 0.9  0.03]
Expected Normalized
[0.71 0.9  0.7 ]
```

[20]:
```
# q gradient (0.8,2)
q_pull_count   = np.zeros((num_ep,num_bandit))
q_estimation   = np.zeros((num_ep,num_bandit))
q_reward       = np.zeros((num_ep,num_iter))
q_optimal_pull = np.zeros((num_ep,num_iter))
q_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 2
```

```
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    q_pull_count[eps,:]   = temp_pull_count
    q_estimation[eps,:]   = temp_estimation
    q_reward[eps,:]       = temp_reward
    q_optimal_pull[eps,:] = temp_optimal_pull
    q_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(q_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(q_estimation.mean(0)-q_estimation.mean(0).min())/
 ↪(q_estimation.mean(0).max()-q_estimation.mean(0).min()) + gt_prob.min()
)
q_estimation = (gt_prob.max()-gt_prob.min())*(q_estimation.mean(0)-q_estimation.
 ↪mean(0).min())/(q_estimation.mean(0).max()-q_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.29 0.3  0.41]
```

```
Expected Normalized
[0.7   0.719 0.9  ]
```

[21]:
```
# r gradient (0.8,5)
r_pull_count   = np.zeros((num_ep,num_bandit))
r_estimation   = np.zeros((num_ep,num_bandit))
r_reward       = np.zeros((num_ep,num_iter))
r_optimal_pull = np.zeros((num_ep,num_iter))
r_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.8
    baseline = 2
    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    r_pull_count[eps,:]   = temp_pull_count
    r_estimation[eps,:]   = temp_estimation
    r_reward[eps,:]       = temp_reward
    r_optimal_pull[eps,:] = temp_optimal_pull
```

```
    r_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(r_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(r_estimation.mean(0)-r_estimation.mean(0).min())/
 ↪(r_estimation.mean(0).max()-r_estimation.mean(0).min()) + gt_prob.min()
)
r_estimation = (gt_prob.max()-gt_prob.min())*(r_estimation.mean(0)-r_estimation.
 ↪mean(0).min())/(r_estimation.mean(0).max()-r_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 0.98  0.08 -0.06]
Expected Normalized
[0.9   0.728 0.7  ]
```

Take the time varying as the variable

Actually, time varying variable is highly connected to the reccommendation system, so there's a lot of implementation of the rule of time variables. We picked the technique of Cumulative Take-Rate replay in order to evaluate the accuracy of the Gradient Bandit algorithm. The function is defined as: $C(T) = \frac{\Sigma_{t=1}^{T} y_t x 1[\pi_{t-1}(x_t)=a_t]}{\Sigma_{t=1}^{T} 1[\pi_{t-1}(x_t)=a_t]}$ Whenever the predicted arm is equal to the current arm, the identity function evaluates to one and CTR is updated for that time stamp.

for α, we picked different representable values. 1. α=1 2. α=0.001 3. $\alpha=\frac{1}{\sqrt{t}}$ (time varying alpha) 4. α=0.001*$Regret\_mean$

time varying data is in the `dataset.txt` In order to implement a contextual bandit algorithm more like in the real case, I used the personalized dataset consisting of 5000 time steos by Criteo, with each step containing the information of current user action. reward and a vector of size 100 denoting the context. The dataset is downlowded from https://github.com/appurwar/Contextual-Bandit-News-Article-Recommendation, and the data is generated by the function $C(T)$, which is a real-world news article reccommendation dataset.

[22]:
```
with open("dataset.txt","r") as file:
    # Strip the new line character from the end of each line
    dataset = [line.strip("\n") for line in file]
```

[23]:
```
data = []
for line in dataset:
    values = line.split()
    values = list(map(int, values))
    for item in values[0:100]:
        data.append(item)
```

[24]:
```
# s gradient (1,C(T))
s_pull_count   = np.zeros((num_ep,num_bandit))
s_estimation   = np.zeros((num_ep,num_bandit))
s_reward       = np.zeros((num_ep,num_iter))
s_optimal_pull = np.zeros((num_ep,num_iter))
s_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 1

    for iter in range(num_iter):
        baseline = data[num_ep*10+num_iter]
        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    s_pull_count[eps,:]   = temp_pull_count
    s_estimation[eps,:]   = temp_estimation
    s_reward[eps,:]       = temp_reward
    s_optimal_pull[eps,:] = temp_optimal_pull
    s_regret_total[eps,:] = temp_regret

print('Ground Truth')
```

```
print(gt_prob)
print('Expected ')
print(np.around(s_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(s_estimation.mean(0)-s_estimation.mean(0).min())/
 ↪(s_estimation.mean(0).max()-s_estimation.mean(0).min()) + gt_prob.min()
)
s_estimation = (gt_prob.max()-gt_prob.min())*(s_estimation.mean(0)-s_estimation.
 ↪mean(0).min())/(s_estimation.mean(0).max()-s_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[ 1.88  0.88 -1.76]
Expected Normalized
[0.9   0.845 0.7  ]
```

[25]:
```
# t gradient (0.001,C(T))
t_pull_count   = np.zeros((num_ep,num_bandit))
t_estimation   = np.zeros((num_ep,num_bandit))
t_reward       = np.zeros((num_ep,num_iter))
t_optimal_pull = np.zeros((num_ep,num_iter))
t_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.001

    for iter in range(num_iter):
        baseline = data[num_ep*10+num_iter]
        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1
```

```
        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                        (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    t_pull_count[eps,:]   = temp_pull_count
    t_estimation[eps,:]   = temp_estimation
    t_reward[eps,:]       = temp_reward
    t_optimal_pull[eps,:] = temp_optimal_pull
    t_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(t_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(t_estimation.mean(0)-t_estimation.mean(0).min())/
↪(t_estimation.mean(0).max()-t_estimation.mean(0).min()) + gt_prob.min()
)
t_estimation = (gt_prob.max()-gt_prob.min())*(t_estimation.mean(0)-t_estimation.
↪mean(0).min())/(t_estimation.mean(0).max()-t_estimation.mean(0).min()) + gt_prob.
↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.38 0.51 0.11]
Expected Normalized
[0.834 0.9   0.7  ]
```

[43]:
```
# u gradient (regret)
u_pull_count   = np.zeros((num_ep,num_bandit))
u_estimation   = np.zeros((num_ep,num_bandit))
u_reward       = np.zeros((num_ep,num_iter))
u_optimal_pull = np.zeros((num_ep,num_iter))
u_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
```

```
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    alpha = 0.001 * temp_mean_reward

    for iter in range(num_iter):
        beta = data[num_ep*10+num_iter]
        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-temp_mean_reward))/
↪(iter) if not iter==0 else ((current_reward-temp_mean_reward))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    u_pull_count[eps,:]   = temp_pull_count
    u_estimation[eps,:]   = temp_estimation
    u_reward[eps,:]       = temp_reward
    u_optimal_pull[eps,:] = temp_optimal_pull
    u_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(u_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(u_estimation.mean(0)-u_estimation.mean(0).min())/
↪(u_estimation.mean(0).max()-u_estimation.mean(0).min()) + gt_prob.min()
)
```

```
u_estimation = (gt_prob.max()-gt_prob.min())*(u_estimation.mean(0)-u_estimation.
 ↪mean(0).min())/(u_estimation.mean(0).max()-u_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.5  0.32 0.17]
Expected Normalized
[0.9   0.792 0.7  ]
```

[48]:
```
# v gradient (1/sqrt(t),C(T))
v_pull_count   = np.zeros((num_ep,num_bandit))
v_estimation   = np.zeros((num_ep,num_bandit))
v_reward       = np.zeros((num_ep,num_iter))
v_optimal_pull = np.zeros((num_ep,num_iter))
v_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    temp_mean_reward = 0
    count = 1
    for iter in range(num_iter):
        alpha = 1/math.sqrt(count)
        baseline = data[num_ep*10+num_iter]
        count += 1
        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation) / np.sum(np.exp(temp_estimation))
        current_choice = np.random.choice(num_bandit,p=pi)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        temp_mean_reward = temp_mean_reward + ((current_reward-baseline))/(iter) if␣
 ↪not iter==0 else ((current_reward-baseline))
        mask = np.zeros(num_bandit)
        mask[current_choice] = 1

        temp_estimation = (mask)   *␣
 ↪(temp_estimation+alpha*(current_reward-temp_mean_reward)*(1-pi)) + \
                          (1-mask) *␣
 ↪(temp_estimation-alpha*(current_reward-temp_mean_reward)*(pi))

        # update reward and optimal choice
```

```
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
 ↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

    v_pull_count[eps,:]   = temp_pull_count
    v_estimation[eps,:]   = temp_estimation
    v_reward[eps,:]       = temp_reward
    v_optimal_pull[eps,:] = temp_optimal_pull
    v_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(v_estimation.mean(0),2))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(v_estimation.mean(0)-v_estimation.mean(0).min())/
 ↪(v_estimation.mean(0).max()-v_estimation.mean(0).min()) + gt_prob.min()
)
v_estimation = (gt_prob.max()-gt_prob.min())*(v_estimation.mean(0)-v_estimation.
 ↪mean(0).min())/(v_estimation.mean(0).max()-v_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.35 0.33 0.32]
Expected Normalized
[0.9   0.774 0.7  ]
```

VII. Report of the above simulation

regret total report

title	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
optimal percentage	0.933638	0.66643	0.400068	0.831268	0.47381	0.471448	0.333108	0.81981	0.33211	0.394824	0.310568	0.31	0.412046	0.434292	0.315352	0.329694	0.377286	0.338454	0.316802	0.362222	0.349356

Total Regret Analysis for Classical Bandit Problems The purpose our algorithm should achieve is to minimize $\text{Regret}(T, \pi, f^*) = \mathbb{E}\left[\sum_{t=1}^{T} f^*(x^*) - f^*(x_t)\right]$, where the $f^* : \mathcal{X} \to \mathbb{R}$ stands for the oracle function convert actions to rewards and $r_t = f^*(x_t) + w_t$ denote the reward gained at time t where w_t is some zero-mean noise

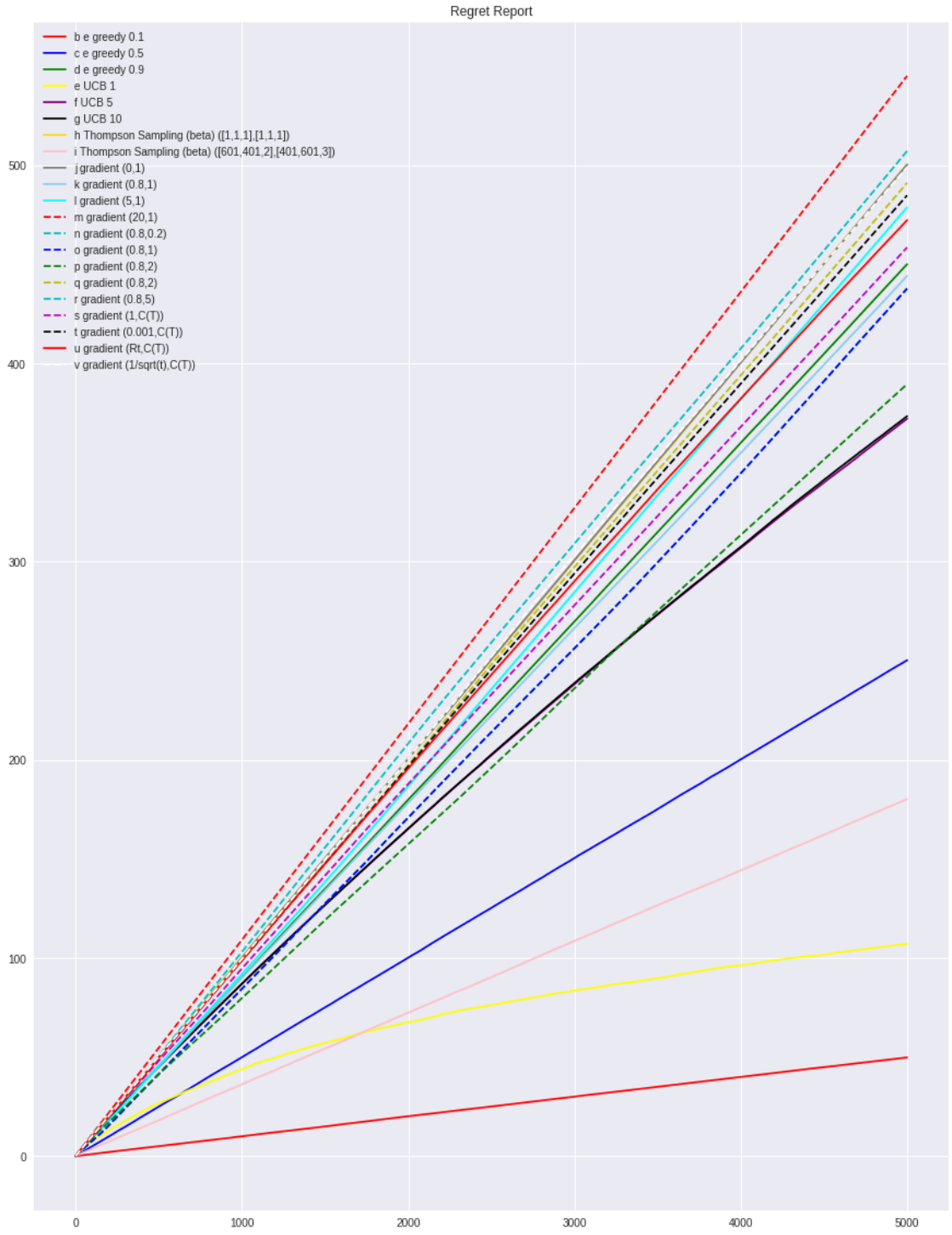
the insights from the report In this report, we can see that most of the case has the 7 digits regret total. In the ϵ*greedy* algorithm, the bigger the ϵ, the bigger the regret total. This may be-

cause small ϵ can have the lower exploration rate, it just almost always pull the optimal situation. As for the UCB algorithm, e UCB 1 has the fewest regret total. this may because parameter $c\geq 0$ $controls the degree of optimism. So the greater the optimism, the more regret it may get. For TS algorithm, it would quickly find the arm with and β_j. For the Gradient bandit algorithm, most of the regret total is highest among 4 algorithms. The total regret of time varying varibles is generally bigger than those of time constant variables.

regret report

[60]:
```
plt.figure(figsize=(15,20))
plt.plot(b_regret_total.mean(0),c='red',    label='b e greedy 0.1')
plt.plot(c_regret_total.mean(0),c='blue',   label='c e greedy 0.5 ')
plt.plot(d_regret_total.mean(0),c='green',  label='d e greedy 0.9')
plt.plot(e_regret_total.mean(0),c='yellow', label='e UCB 1')
plt.plot(f_regret_total.mean(0),c='purple', label='f UCB 5')
plt.plot(g_regret_total.mean(0),c='black',  label='g UCB 10')
plt.plot(h_regret_total.mean(0),c='gold',   label='h Thompson Sampling (beta)␣
 ↪([1,1,1],[1,1,1])')
plt.plot(i_regret_total.mean(0),c='pink',   label='i Thompson Sampling (beta)␣
 ↪([601,401,2],[401,601,3])')
plt.plot(j_regret_total.mean(0),c='grey',   label='j gradient (0,1)')
plt.plot(k_regret_total.mean(0),c='skyblue',label='k gradient (0.8,1)')
plt.plot(l_regret_total.mean(0),c='cyan',   label='l gradient (5,1)')
plt.plot(m_regret_total.mean(0),'r--',label='m gradient (20,1)')
plt.plot(n_regret_total.mean(0),'c--',label='n gradient (0.8,0.2)')
plt.plot(n_regret_total.mean(0),'b--',label='o gradient (0.8,1)')
plt.plot(o_regret_total.mean(0),'g--',label='p gradient (0.8,2)')
plt.plot(p_regret_total.mean(0),'y--',label='q gradient (0.8,2)')
plt.plot(q_regret_total.mean(0),'c--',label='r gradient (0.8,5)')
plt.plot(r_regret_total.mean(0),'m--',label='s gradient (1,C(T))')
plt.plot(s_regret_total.mean(0),'k--',label='t gradient (0.001,C(T))')
plt.plot(u_regret_total.mean(0),'r',label='u gradient (Rt,C(T))')
plt.plot(t_regret_total.mean(0),'w--',label='v gradient (1/sqrt(t),C(T))')

plt.title("Regret Report")
plt.legend(prop={'size': 10})
plt.show()
```

Problem 2 continued on next page...



Regret Analysis for Classical Bandit Problems **Asymptotic Instance Dependent Regret Bounds.** Sharp results on the asymptotic scaling of regret are available. The cumulative regret of an algorithm over T periods is $Regret(T) = \Sigma_{t=1}^{T}(\max_{1\leq k\leq K}\theta_k - \theta_{x_t})$, where K represent the number of actions. For each time horizon T, $\mathscr{E}[Regret(T)|\theta]$ measures the expected T-period regret on the problem instance θ. The conditional expecta-

Problem 2 continued on next page...

tion integrates over the noisy realizations of rewards and the algorithm's random action selection, holding fixed the success probabilities $\theta = (\theta_1, ..., \theta_K)$. Though this is difficult to evaluate, one can show that

$$lim_{T->\infty} \frac{\mathscr{E}[Regret(T)|\theta]}{log(T)} = \Sigma_{k \neq k^*} \frac{\theta_{k^*} - \theta_k}{d_{KL}(\theta_{k^*}||\theta_k)}$$

assuming that there is a unique optimal action k^*. Here, $d_{KL}(\theta||\theta') = \theta log(\frac{\theta}{\theta'}) + (1-\theta) log(\frac{1-\theta}{1-\theta'})$ is the KL divergence.

the insights from the graph In this report, `m gradient(0,1)` has the biggest regret with the time of pulls and b e greedy 0.1 has the flattest slope. All other lines are in between. Also, one of another noticable phenomenon is between line p and line c. Initially, `c greedy 0.5` first grow faster than `p gradient(0.8,2)`. However, `p` has steeper slope after arontd 3400 pulls. From this graph we can conclude that `u` may have the best performance due to the regret has the inverse relationship with the bandit performance.

The percentage of plays in which the optimal arm is pulled

title	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
optimal percentage	0.933638	0.66643	0.400068	0.831268	0.47381	0.471448	0.333108	0.81981	0.33211	0.394824	0.310568	0.31	0.412046	0.434292	0.315352	0.329694	0.377286	0.338454	0.316802	0.362222	0.349356

the insight from the report As we can see, `b e greedy 0.1` has the best optimal percentage, which is in accordance with the total regret report. And parameter ϵ stands for the the rate of explotation.

Also, the trend of paramters in UCB algorithm is basically the same as the tootal regret report.

Let's talk a little bit about the TS algorithm, Since these estimates are drawn from posterior distributions, each of these probabilities is also equal to the probability that the corresponding action is optimal, conditioned on observed history.

The algorithm with baseline in Gradient Bandit is better than those without baseline.

An important innovation in Gradient Bandit is the definition of a "preference" function for actions H_t, representing the degree of preference for action a. Use this preference function to calculate the probability of selecting an action. Of course, the stronger the preference, the greater the probability of choosing the action

From the report, we can conclude that the time varying Gradient Bandit have more explotation rate because they gnerally has fewer optimal percentage.

VIII. Comparison of 4 bandit algorithms and their parameters

The gaps between the algorithm outputs and the oracle value

title	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
gap_1	0.0005	-1.672e-05	-0.001	-0.0001	-0.0001	0.0011	0.4003	0.2988	0.0860	0.0860	0.1141	0.1212	0.1086	0.1043	0.1301	0.1270	0.1241	0.0850	0.0885	0.1027	0.1085
gap_2	0.00373	-0.0007	0.0021	0.0046	0.0017	0.00013	0.33	0.400	-0.01	-0.0139	0.0141	0.021	0.008	0.0043	0.0301	0.0270	0.0241	-0.0149	-0.0114	0.0027	0.008
gap_3	0.00849	0.00028	0.00075	0.01600	0.00149	0.00148	0.23539	0.25709	-0.1124	-0.1139	-0.0858	-0.0787	-0.0913	-0.0956	-0.0698	-0.0729	-0.0758	-0.1149	-0.1114	-0.0972	-0.0914

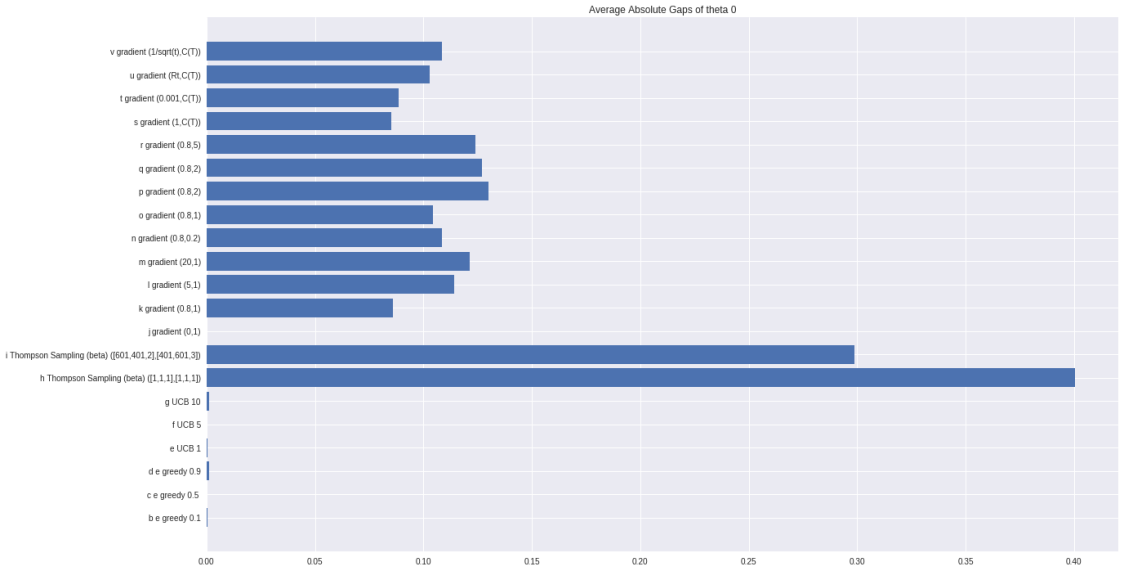
Problem 2 continued on next page...

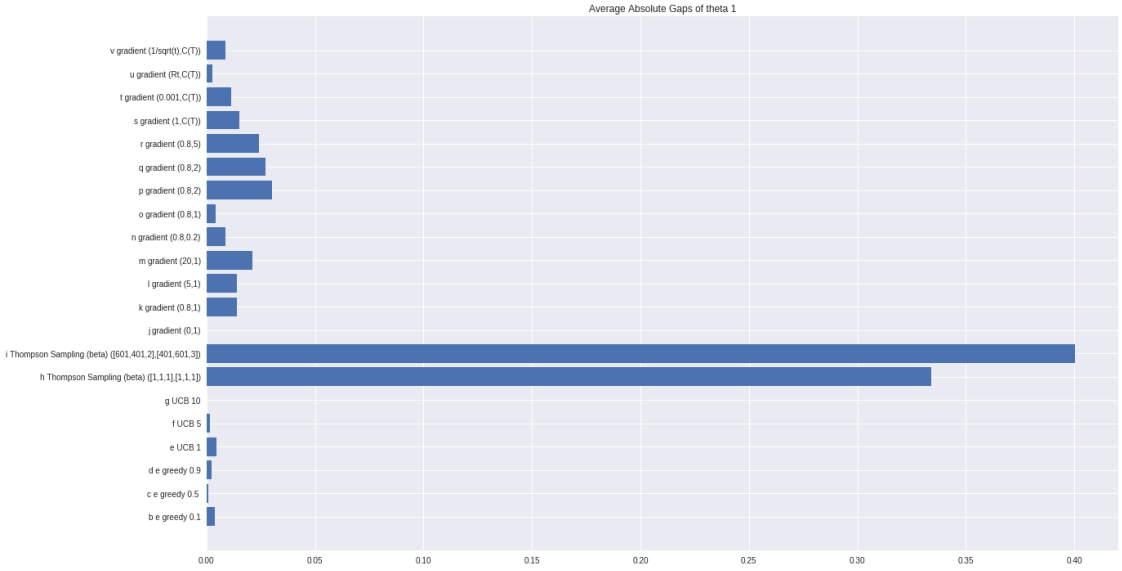
The sum of absolute gaps between the algorithm outputs and the oracle value

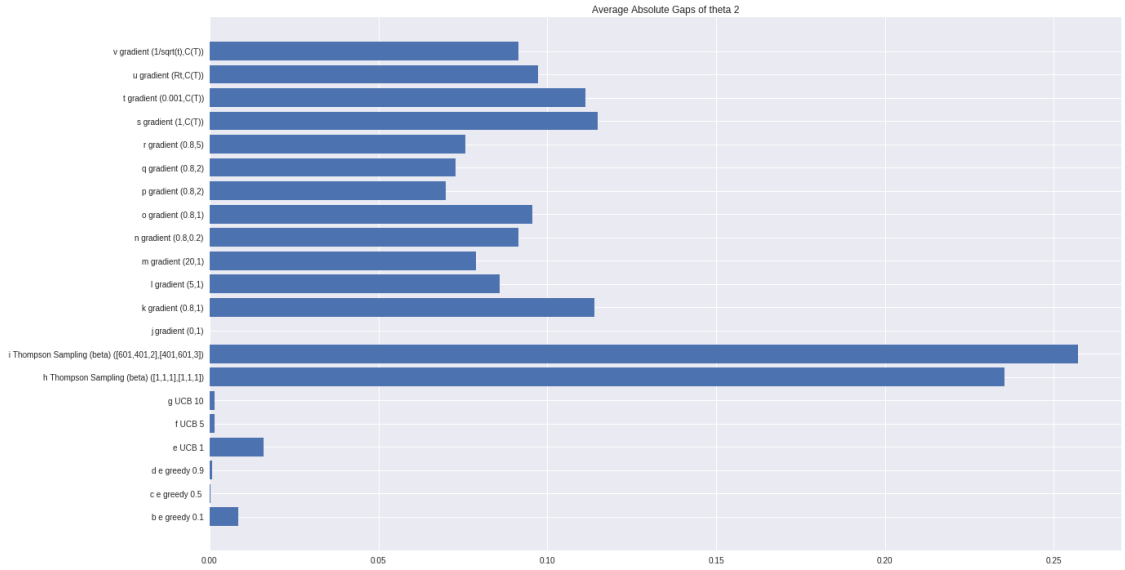
[59]:
```
for i in range(3):
    plt.figure(figsize=(20, 12))
    plt.title("Average Absolute Gaps of theta {}".format(i))
    plt.barh([
"b e greedy 0.1",
"c e greedy 0.5 ",
"d e greedy 0.9",
"e UCB 1",
"f UCB 5",
"g UCB 10",
"h Thompson Sampling (beta) ([1,1,1],[1,1,1])",
"i Thompson Sampling (beta) ([601,401,2],[401,601,3])",
"j gradient (0,1)",
"k gradient (0.8,1)",
"l gradient (5,1)",
"m gradient (20,1)",
"n gradient (0.8,0.2)",
"o gradient (0.8,1)",
"p gradient (0.8,2)",
"q gradient (0.8,2)",
"r gradient (0.8,5)",
"s gradient (1,C(T))",
"t gradient (0.001,C(T))",
"u gradient (Rt,C(T))",
"v gradient (1/sqrt(t),C(T))",
    ], [
        np.abs(gt_prob-b_estimation.mean(0))[i],
        np.abs(gt_prob-c_estimation.mean(0))[i],
        np.abs(gt_prob-d_estimation.mean(0))[i],
        np.abs(gt_prob-e_estimation.mean(0))[i],
        np.abs(gt_prob-f_estimation.mean(0))[i],
        np.abs(gt_prob-g_estimation.mean(0))[i],
        np.abs(gt_prob-h_estimation.mean(0))[i],
        np.abs(gt_prob-i_estimation.mean(0))[i],
        np.abs(gt_prob-j_estimation.mean(0))[i],
        np.abs(gt_prob-k_estimation.mean(0))[i],
        np.abs(gt_prob-l_estimation.mean(0))[i],
        np.abs(gt_prob-m_estimation.mean(0))[i],
        np.abs(gt_prob-n_estimation.mean(0))[i],
        np.abs(gt_prob-o_estimation.mean(0))[i],
        np.abs(gt_prob-p_estimation.mean(0))[i],
        np.abs(gt_prob-q_estimation.mean(0))[i],
        np.abs(gt_prob-r_estimation.mean(0))[i],
        np.abs(gt_prob-s_estimation.mean(0))[i],
        np.abs(gt_prob-t_estimation.mean(0))[i],
        np.abs(gt_prob-u_estimation.mean(0))[i],
        np.abs(gt_prob-v_estimation.mean(0))[i],
```

Problem 2 continued on next page...

```
])
plt.show()
```


Average Absolute Gaps of theta 0
v gradient (1/sqrt(t),C(T))
u gradient (Rt,C(T))
t gradient (0.001,C(T))
s gradient (1,C(T))
r gradient (0.8,5)
q gradient (0.8,2)
p gradient (0.8,2)
o gradient (0.8,1)
n gradient (0.8,0.2)
m gradient (20,1)
l gradient (5,1)
k gradient (0.8,1)
j gradient (0,1)
i Thompson Sampling (beta) ([601,401,2],[401,601,3])
h Thompson Sampling (beta) ([1,1,1],[1,1,1])
g UCB 10
f UCB 5
e UCB 1
d e greedy 0.9
c e greedy 0.5
b e greedy 0.1
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40


Average Absolute Gaps of theta 1
v gradient (1/sqrt(t),C(T))
u gradient (Rt,C(T))
t gradient (0.001,C(T))
s gradient (1,C(T))
r gradient (0.8,5)
q gradient (0.8,2)
p gradient (0.8,2)
o gradient (0.8,1)
n gradient (0.8,0.2)
m gradient (20,1)
l gradient (5,1)
k gradient (0.8,1)
j gradient (0,1)
i Thompson Sampling (beta) ([601,401,2],[401,601,3])
h Thompson Sampling (beta) ([1,1,1],[1,1,1])
g UCB 10
f UCB 5
e UCB 1
d e greedy 0.9
c e greedy 0.5
b e greedy 0.1
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40



A brief summary of 4 simulations with regard of total average absolute gaps

In the perspective of the total average absolute gaps, ϵ-greedy algorithm runs the best, which means ϵ-greedy algorithm predicts the oracle θ_js the best.

However, in the perspective of total average rewards, Thompson Sampling algorithm runs the best, which means Thompson Sampling gets the most reward as a character of gambler.

Also, we can see that: for θ_0 and θ_1, ϵ *greedy* algorithm gives almost the most precise predictions and the predictions from Thompson Sampling algorithm are very rough; but for θ_2, Thompson Sampling algorithm gives the prediction equally precise with other algorithms.

And as for the gradient bandit doesn't show

A brief explanation of the phenomenon ϵ-greedy algorithm will always have a chance to randomly pick bandit arms, so its predictions of θ_js could be equally precise.

However, Thompson Sampling algorithm tends to always pick the arm with the highest predicted θ_j after a little rounds of testing, therefore it gives rough prediction of θ_0 and θ_1, but a much preciser prediction of θ_2.

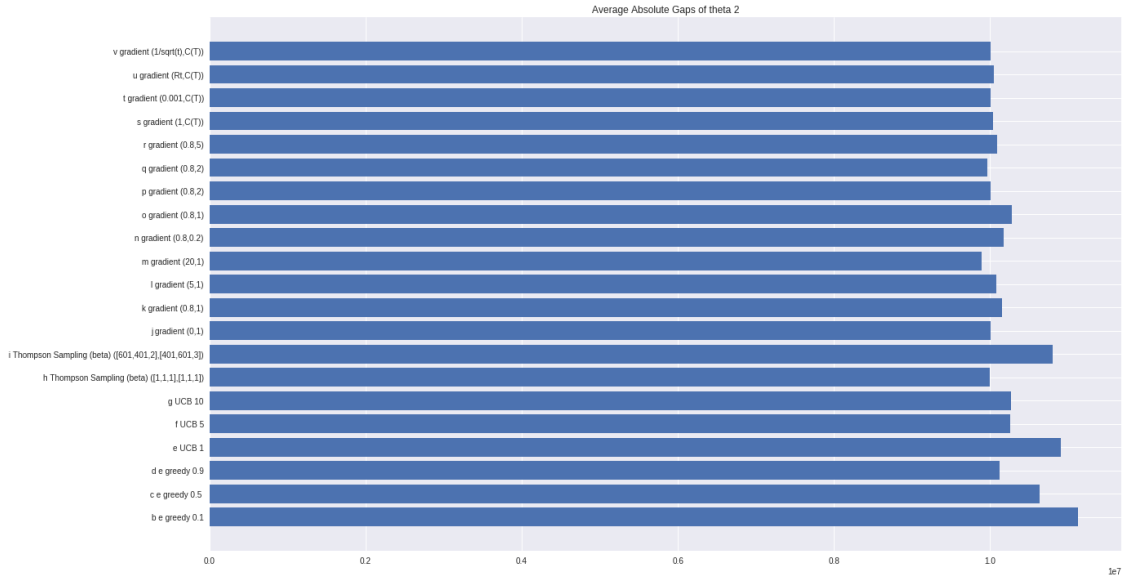
For UCB algorithm, it sythesize strategies that to exploit the arm with the highest predicted θ_j and to explore arms with little picks. Therefore, the performance of UCB algorithm tends to be in the middle of ϵ-greedy algorithm and Thompson Sampling algorithm.

Also, the Bandit gradient, `v gradient` which is the time varying is slightly better than ϵ *greedy* in that case.

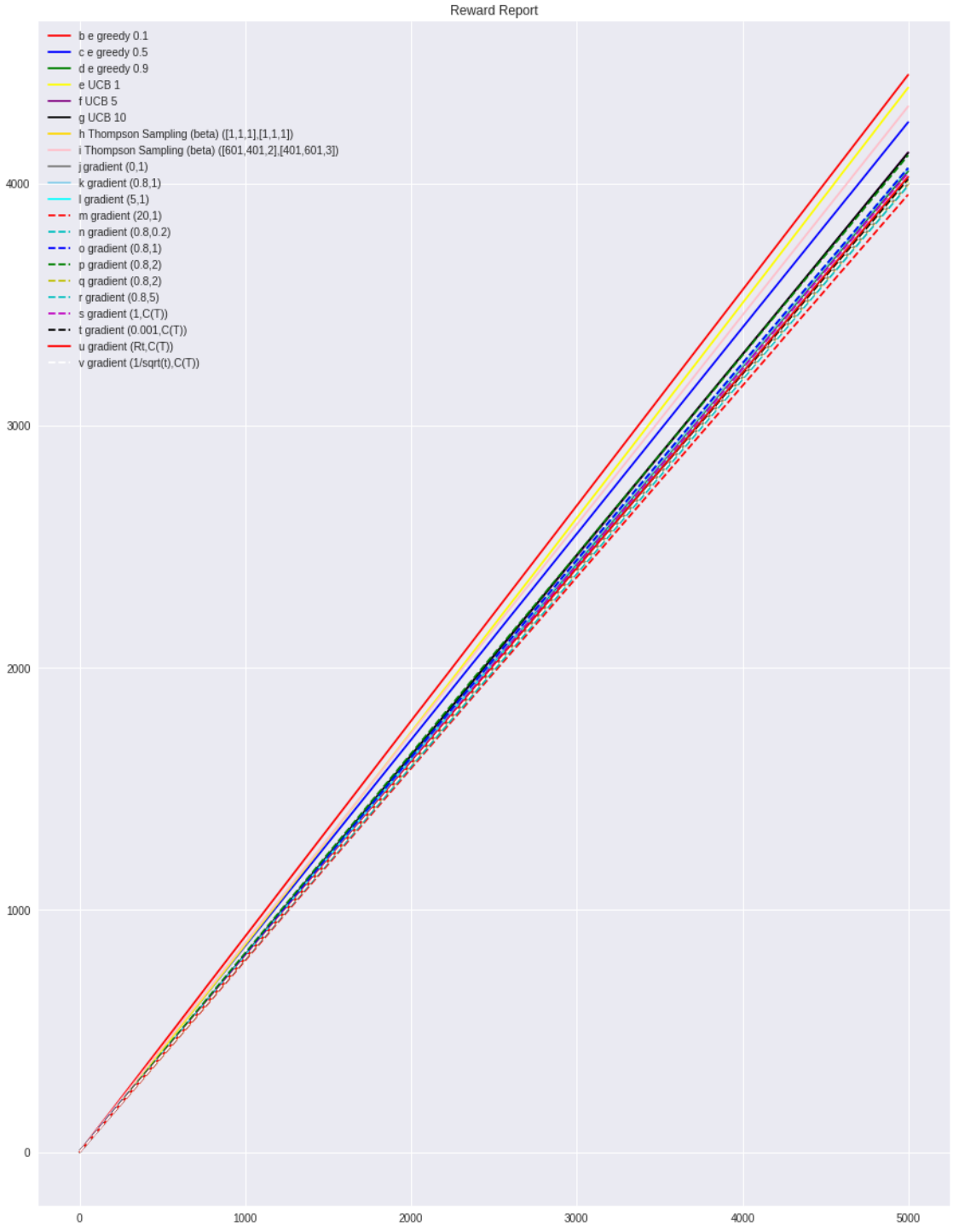
The difference between the algorithm reward

[65]:
```
plt.figure(figsize=(20, 12))
plt.title("Average Absolute Gaps of theta {}".format(i))
plt.barh([
```

```
"b e greedy 0.1",
"c e greedy 0.5 ",
"d e greedy 0.9",
"e UCB 1",
"f UCB 5",
"g UCB 10",
"h Thompson Sampling (beta) ([1,1,1],[1,1,1])",
"i Thompson Sampling (beta) ([601,401,2],[401,601,3])",
"j gradient (0,1)",
"k gradient (0.8,1)",
"l gradient (5,1)",
"m gradient (20,1)",
"n gradient (0.8,0.2)",
"o gradient (0.8,1)",
"p gradient (0.8,2)",
"q gradient (0.8,2)",
"r gradient (0.8,5)",
"s gradient (1,C(T))",
"t gradient (0.001,C(T))",
"u gradient (Rt,C(T))",
"v gradient (1/sqrt(t),C(T))",

], [
        np.abs(b_reward.mean(0).sum()),
        np.abs(c_reward.mean(0).sum()),
        np.abs(d_reward.mean(0).sum()),
        np.abs(e_reward.mean(0).sum()),
        np.abs(f_reward.mean(0).sum()),
        np.abs(g_reward.mean(0).sum()),
        np.abs(h_reward.mean(0).sum()),
        np.abs(i_reward.mean(0).sum()),
        np.abs(j_reward.mean(0).sum()),
        np.abs(k_reward.mean(0).sum()),
        np.abs(l_reward.mean(0).sum()),
        np.abs(m_reward.mean(0).sum()),
        np.abs(n_reward.mean(0).sum()),
        np.abs(o_reward.mean(0).sum()),
        np.abs(p_reward.mean(0).sum()),
        np.abs(q_reward.mean(0).sum()),
        np.abs(r_reward.mean(0).sum()),
        np.abs(s_reward.mean(0).sum()),
        np.abs(t_reward.mean(0).sum()),
        np.abs(u_reward.mean(0).sum()),
        np.abs(v_reward.mean(0).sum()),

])
plt.show()
```

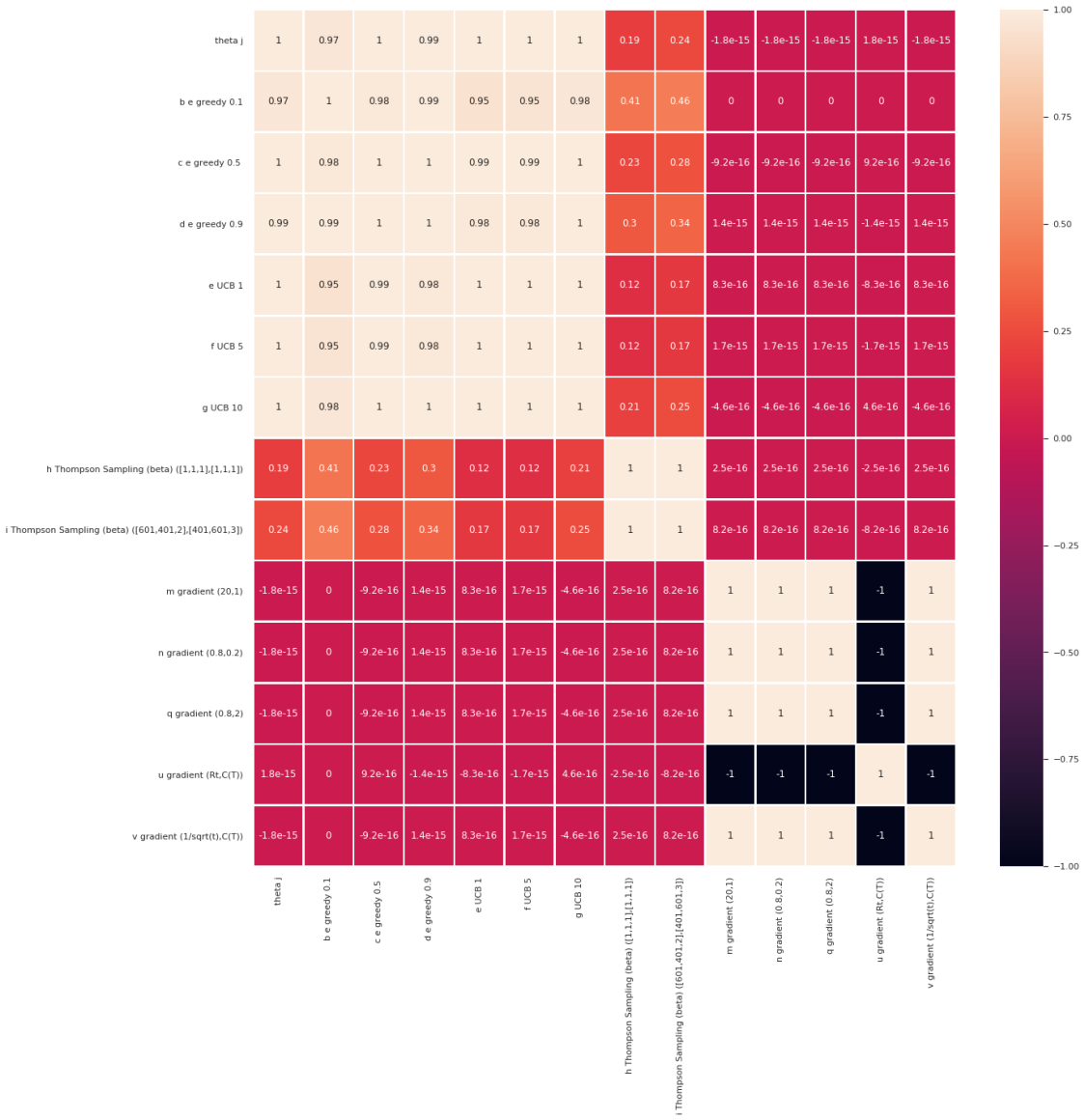


[68]:
```
plt.figure(figsize=(15,20))
plt.plot(b_reward.mean(0),c='red',    label='b e greedy 0.1')
plt.plot(c_reward.mean(0),c='blue',   label='c e greedy 0.5 ')
plt.plot(d_reward.mean(0),c='green',  label='d e greedy 0.9')
plt.plot(e_reward.mean(0),c='yellow', label='e UCB 1')
plt.plot(f_reward.mean(0),c='purple', label='f UCB 5')
plt.plot(g_reward.mean(0),c='black',  label='g UCB 10')
plt.plot(h_reward.mean(0),c='gold',   label='h Thompson Sampling (beta)␣
 ↪([1,1,1],[1,1,1])')
plt.plot(i_reward.mean(0),c='pink',   label='i Thompson Sampling (beta)␣
 ↪([601,401,2],[401,601,3])')
plt.plot(j_reward.mean(0),c='grey',   label='j gradient (0,1)')
plt.plot(k_reward.mean(0),c='skyblue',label='k gradient (0.8,1)')
plt.plot(l_reward.mean(0),c='cyan',   label='l gradient (5,1)')
plt.plot(m_reward.mean(0),'r--',label='m gradient (20,1)')
plt.plot(n_reward.mean(0),'c--',label='n gradient (0.8,0.2)')
plt.plot(n_reward.mean(0),'b--',label='o gradient (0.8,1)')
plt.plot(o_reward.mean(0),'g--',label='p gradient (0.8,2)')
plt.plot(p_reward.mean(0),'y--',label='q gradient (0.8,2)')
plt.plot(q_reward.mean(0),'c--',label='r gradient (0.8,5)')
plt.plot(r_reward.mean(0),'m--',label='s gradient (1,C(T))')
plt.plot(s_reward.mean(0),'k--',label='t gradient (0.001,C(T))')
plt.plot(u_reward.mean(0),'r',label='u gradient (Rt,C(T))')
plt.plot(t_reward.mean(0),'w--',label='v gradient (1/sqrt(t),C(T))')

plt.title("Reward Report")
plt.legend(prop={'size': 10})
plt.show()
```



[77]:
```
# correaltion
import pandas as pd
import seaborn as sns; sns.set()
data = pd.DataFrame({
```

```
        'theta j':gt_prob,
        'b e greedy 0.1':b_estimation[1],
        'c e greedy 0.5 ':c_estimation[1],
        'd e greedy 0.9':d_estimation[1],
        'e UCB 1':e_estimation[1],
        'f UCB 5':f_estimation[1],
        'g UCB 10':g_estimation[1],
        'h Thompson Sampling (beta) ([1,1,1],[1,1,1])':h_estimation[1],
        'i Thompson Sampling (beta) ([601,401,2],[401,601,3])':i_estimation[1],
        'm gradient (20,1)':m_estimation[1],
        'n gradient (0.8,0.2)':n_estimation[1],
        'q gradient (0.8,2)':q_estimation[1],
        'u gradient (Rt,C(T))':u_estimation[1],
        'v gradient (1/sqrt(t),C(T))':v_estimation[1],
    })
plt.figure(figsize=(20,20))
sns.heatmap(data.corr(),annot=True,linewidths=1.5)
plt.show()
```



correaltion comparison

However, when we view the correlation matrix between the ground truth probability distribution, we can see that e-greedy and gradient bandit had one to one correlation.

Quite a surprising result can be seen above, we can notice right away that simpler methods such as greedy methods are able to outperform more advanced solutions.

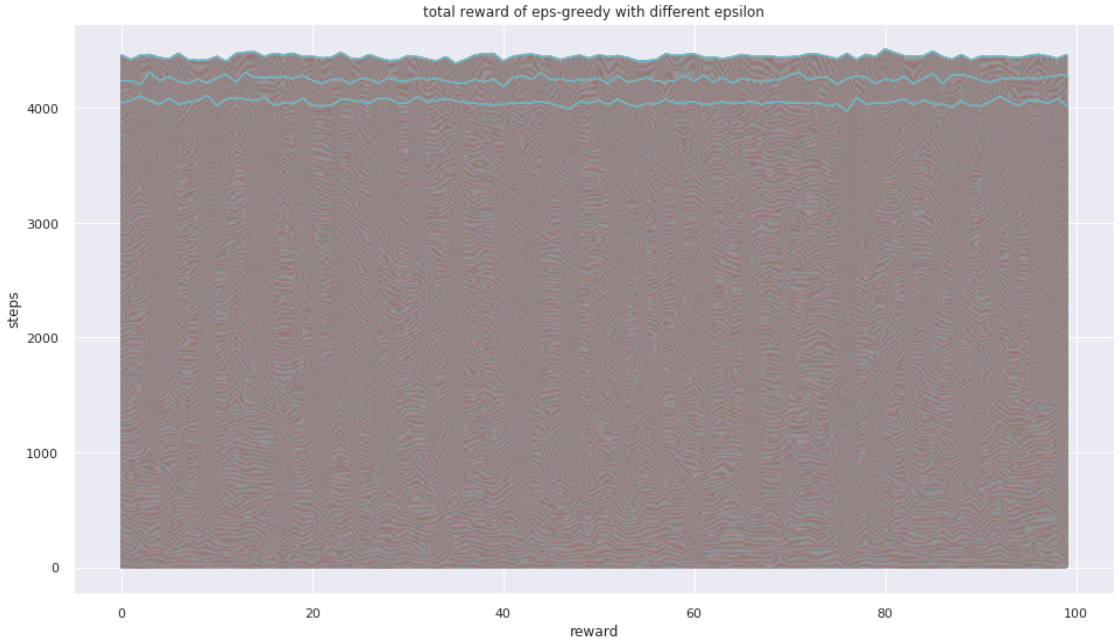
We can see that the `u Gradient(Rt,C(T))` and `e ucb 1` method gave us the most accurate estimations.

The impact of the ϵ for ϵ*greedy* algorithm

[89]:
```
plt.figure(figsize=(16, 9))
plt.plot(b_reward)
```

```
plt.plot(c_reward)
plt.plot(d_reward)
plt.xlabel("reward")
plt.ylabel("steps")
plt.title("total reward of eps-greedy with different epsilon")
```

[89]: `Text(0.5, 1.0, 'total reward of eps-greedy with different epsilon')`

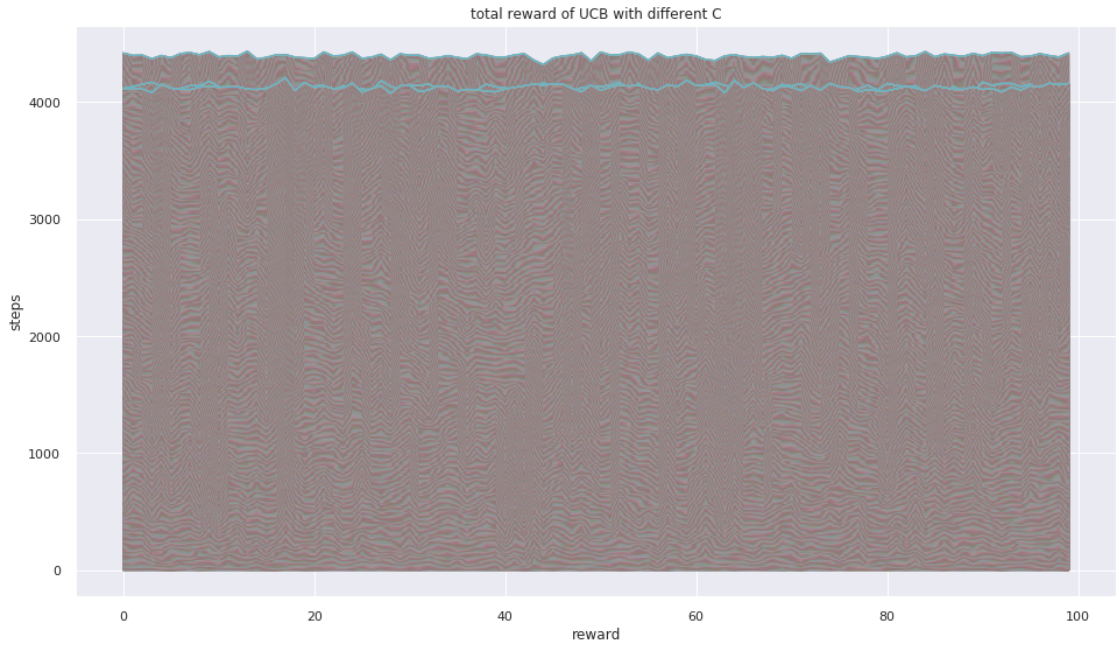


Parameter ϵ is the probability of randomly choosing bandit arm ignoring current predictions. So increase ϵ means more exploration and less expoiltation, which may lead to preciser predictions of θ_js but less total rewards. Therefore, increase ϵ means more exploration and less expoiltation, which may lead to preciser predictions of θ_js but less total rewards. When $\epsilon = 1$, the algorithm deteriorates to random selections. For example, When $\epsilon = 1$, the algorithm deteriorates to random selections. $ Estimate _t = E [f^* | H_t]Pick randomly $x_t \in \mathcal{X}$ with probability ϵ or pick $x_t \in \arg\max_x \hat{f}_t$, which does not work efficiently and is wasteful on exploration because of lack of learning plan.

The impact of the C for UCB algorithm

[90]:
```
plt.figure(figsize=(16, 9))
plt.plot(e_reward)
plt.plot(f_reward)
plt.plot(g_reward)
plt.xlabel("reward")
plt.ylabel("steps")
plt.title("total reward of UCB with different C")
```

[90]: `Text(0.5, 1.0, 'total reward of UCB with different C')`



Parameter c is the weight of the "bonus", which is basicly the standard deviation of the prediction mean. As we exploit on an arm more, the stanstandard deviation of the prediction mean would decrease. Therefore, parameter c actually controls the balance of exploration and exploitation. With higher c, the algorithm tends to explore more and exploit less, which turns out preciser predictions but maybe less total rewards; with lower c, the algorithm tends to explore less and exploit more, which turns out less precise predictions but maybe more total rewards. Mathematically, Estimate a set F_t with high probability that $f^* \in \mathcal{F}_t$ given H_t Pick$x_t \in$ arg
The exploration and exploitation of specific algorithm is discussed.

Exploration gives more information of the model, and exploitation uses this information to construct current optimized strategy. Therefore, exploration tends to happen more in the early stage and exploitation tends to happen more after a number of rounds.

The trade-off between exploration and exploitation is that: if we use more exploration strategy in algorithms, it may have higher cost and slower convergence speed; if we use more exploitation strategy in algorithms, it may lead us to local optimum but not global optimum.

Therefore, we need to design exploration ways that use the lowest cost to reach the most valuable information, and balance exploration and exploitation dynamicly based on current information.

Multi-armed Bandit with Dependent Rewards

In this case, rewards are not only influenced by the current choice, but also historical choices. Therefore, every action(pull arm) is contextual and is based on a specific state.

A naive algorithm

Track some fixed amounts of historical choices as the current state, and based on that state to determine next action.

Specifically, suppose the original bandit has n arms and assume we track historical choices of length l, we can generate n^l sequences of actions by permutation with replacement. Then we can construct a super bandit with 2^l arms, each of which represents a sequence of actions of the original bandit and has its own θ, and assume its rewards are independent. Then we run algorithms like mantaining explicit policy over the bandits and update them using empirical means. In the beginning, the probability distribution is initialized uniformly (meaning each arm has an equal chance of getting picked) but over time the distribution changes. To optimize, we may drop some arms in the super bandit during the algorithm if their θs are too low, which means such sequences of actions are worthless to try; or combine some arms in the super bandit during the algorithm if their θs always seem to be equal after many rounds and they are the permutations of same actions.

[98]:
```
# x naive algorithm
x_pull_count   = np.zeros((num_ep,num_bandit))
x_estimation   = np.zeros((num_ep,num_bandit))
x_reward       = np.zeros((num_ep,num_iter))
x_optimal_pull = np.zeros((num_ep,num_iter))
x_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    learning_rate = 0.1
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1.0/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        current_choice = np.random.choice(num_bandit, p=temp_estimation)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        mask = np.zeros(num_bandit)
        mask[current_choice] = 1.0

        if current_reward == 1.0:
            temp_estimation = (mask) * (temp_estimation + learning_rate *␣
↪(1-temp_estimation)) + (1-mask) * ( (1-learning_rate) * temp_estimation)

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])
```

Problem 2 continued on next page...

```
    x_pull_count[eps,:]   = temp_pull_count
    x_estimation[eps,:]   = temp_estimation
    x_reward[eps,:]       = temp_reward
    x_optimal_pull[eps,:] = temp_optimal_pull
    x_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(np.around(x_estimation.mean(0),3))
print('Expected Normalized')
print(
    (gt_prob.max()-gt_prob.min())*(x_estimation.mean(0)-x_estimation.mean(0).min())/
 ↪(x_estimation.mean(0).max()-x_estimation.mean(0).min()) + gt_prob.min()
)
x_estimation = (gt_prob.max()-gt_prob.min())*(x_estimation.mean(0)-x_estimation.
 ↪mean(0).min())/(x_estimation.mean(0).max()-x_estimation.mean(0).min()) + gt_prob.
 ↪min()
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.74 0.18 0.08]
Expected Normalized
[0.9  0.73 0.7 ]
```

The Softmax algorithm

The Softmax algorithm compromises exploration and utilization based on the current average reward value of each action. The Softmax function converts a set of values into a set of probabilities. The larger the value, the higher the corresponding probability, so the higher the current average reward value. The greater the chance that the action is selected. The Softmax equation is defined as $P(k) = \frac{e^{\frac{Q(k)}{\gamma}}}{\Sigma_{i=1}^{K} e^{\frac{Q(i)}{\gamma}}}$, T is temperature, which means when it is getting close to 0, the distance is getting bigger, vice versa.

Specifically, we have the algorithm below:

Algorithm 2: Softmax in 3-armed Bernoulli bandit problem

Initialize: $T,\ k = 0, r = 0, Q(i) = 0\ for\ i \in \{1, 2, 3\}, Reward\ function\ R(t)$

```
#Sample model
for t=1,2...,N do
    k ← random(1, 3)
    Sample v from R(k)
    r ← r + v
    Q(k) ← (Q(k)*count(k)+v) / (count(k)+1)
    Baseline ← b
```

[96]:
```
# w softmax
w_pull_count   = np.zeros((num_ep,num_bandit))
w_estimation   = np.zeros((num_ep,num_bandit))
w_reward       = np.zeros((num_ep,num_iter))
w_optimal_pull = np.zeros((num_ep,num_iter))
w_regret_total = np.zeros((num_ep,num_iter))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
    temp_estimation   = np.zeros(num_bandit) + 1/num_bandit
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)
    temp_regret = np.zeros(num_iter)
    tempture = 30

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        pi  = np.exp(temp_estimation/tempture) / np.sum(np.exp(temp_estimation/
↪tempture))
        current_choice = np.random.choice(num_bandit)
        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1
        temp_estimation[current_choice] = temp_estimation[current_choice] + (1/
↪(temp_pull_count[current_choice]+1)) *␣
↪(current_reward-temp_estimation[current_choice])

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
```

```
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter
 ↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])

        # decay the temp
        tempture = tempture * 0.99

    w_pull_count[eps,:]   = temp_pull_count
    w_estimation[eps,:]   = temp_estimation
    w_reward[eps,:]       = temp_reward
    w_optimal_pull[eps,:] = temp_optimal_pull
    w_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(w_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.9   0.8   0.701]
```

Similarly, most of the dependent algorithm can use the above algorithm, just change the $Q(i)$ into State value function (V) and continually use Boltzmann Exploration or adopt other update function, we can get observable Markov decision process (POMDP). Those algorithm we don't consider here.

More advanced, we may use some machine learning algorithm (Neural Networks/Reinforced Learning) to solve this model.

Neural Network

we use partially UCB and partially decay epsilon to model this problem, and dynamicly add/deleta states based on current information to optimize. Then we use the neural network to learn from the current state to solve this model. Notice: the neural network is with adam.

[99]:
```
# y neural network (with adam)
y_pull_count   = np.zeros((num_ep,num_bandit))
y_estimation   = np.zeros((num_ep,num_bandit))
y_reward       = np.zeros((num_ep,num_iter))
y_optimal_pull = np.zeros((num_ep,num_iter))
y_regret_total = np.zeros((num_ep,num_iter))

def sigmoid(x): return 1/(1+np.exp(-x))
def d_sigmoid(x): return sigmoid(x)*(1-sigmoid(x))

for eps in range(num_ep):
    temp_pull_count   = np.zeros(num_bandit)
```

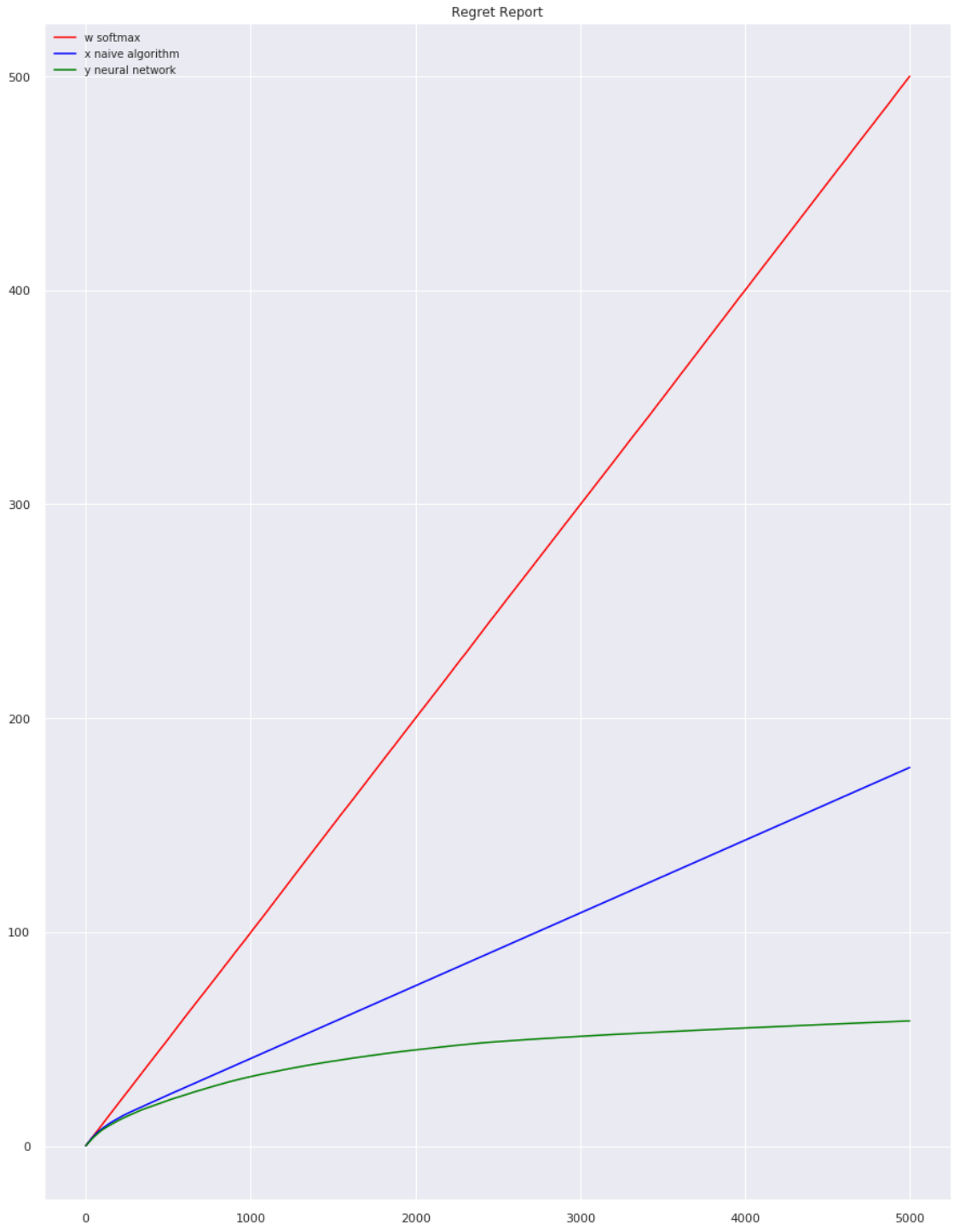
```
    temp_estimation   = np.zeros(num_bandit)
    temp_reward       = np.zeros(num_iter)
    temp_optimal_pull = np.zeros(num_iter)

    weights = np.random.randn(num_bandit,1)
    moment  = np.zeros_like(weights);
    velocity = np.zeros_like(weights);
    epsilon  = 0.3

    for iter in range(num_iter):

        # select bandit / get reward /increase count / update estimate
        if np.random.uniform(0,1)>epsilon:
            current_choice = np.argmax(weights)
            current_input  = np.zeros((1,num_bandit))
            current_input[0,current_choice] = 1
        else:
            current_choice = np.random.choice(np.arange(num_bandit))
            current_input  = np.zeros((1,num_bandit))
            current_input[0,current_choice] = 1

        layer1 = current_input @ weights
        layer1a= sigmoid(layer1)

        current_reward = 1 if np.random.uniform(0,1) < gt_prob[current_choice] else 0
        temp_estimation[current_choice] = temp_estimation[current_choice] +␣
↪current_reward
        temp_pull_count[current_choice] = temp_pull_count[current_choice] + 1

        grad3 = np.log(layer1a+0.0000001) - np.log(temp_estimation[current_choice]/
↪(temp_pull_count[current_choice])+0.0000001)
        grad2 = d_sigmoid(layer1)
        grad1 = current_input
        grad  = grad1.T @ (grad3 * grad2)

        moment   = 0.9*moment + (1-0.9) * grad
        velocity = 0.999*velocity + (1-0.999) * grad**2
        moment_hat   = moment/(1-0.9)
        velocity_hat = velocity/(1-0.999)
        weights  = weights - 0.08 * (moment_hat/(np.sqrt(velocity_hat)+1e-8))

        # update reward and optimal choice
        temp_reward[iter] = current_reward if iter == 0 else temp_reward[iter-1] +␣
↪current_reward
        temp_optimal_pull[iter] = 1 if current_choice == optimal_choice else 0
        temp_regret[iter] = gt_prob[optimal_choice] - gt_prob[current_choice] if iter␣
↪== 0 else temp_regret[iter-1] + (gt_prob[optimal_choice] - gt_prob[current_choice])
```

```
        # Decay the learning rate
        epsilon = epsilon * 0.999

    y_pull_count[eps,:]   = temp_pull_count
    y_estimation[eps,:]   = np.squeeze(sigmoid(weights))
    y_reward[eps,:]       = temp_reward
    y_optimal_pull[eps,:] = temp_optimal_pull
    y_regret_total[eps,:] = temp_regret

print('Ground Truth')
print(gt_prob)
print('Expected ')
print(y_estimation.mean(0))
```

```
Ground Truth
[0.9 0.8 0.7]
Expected
[0.896 0.791 0.696]
```

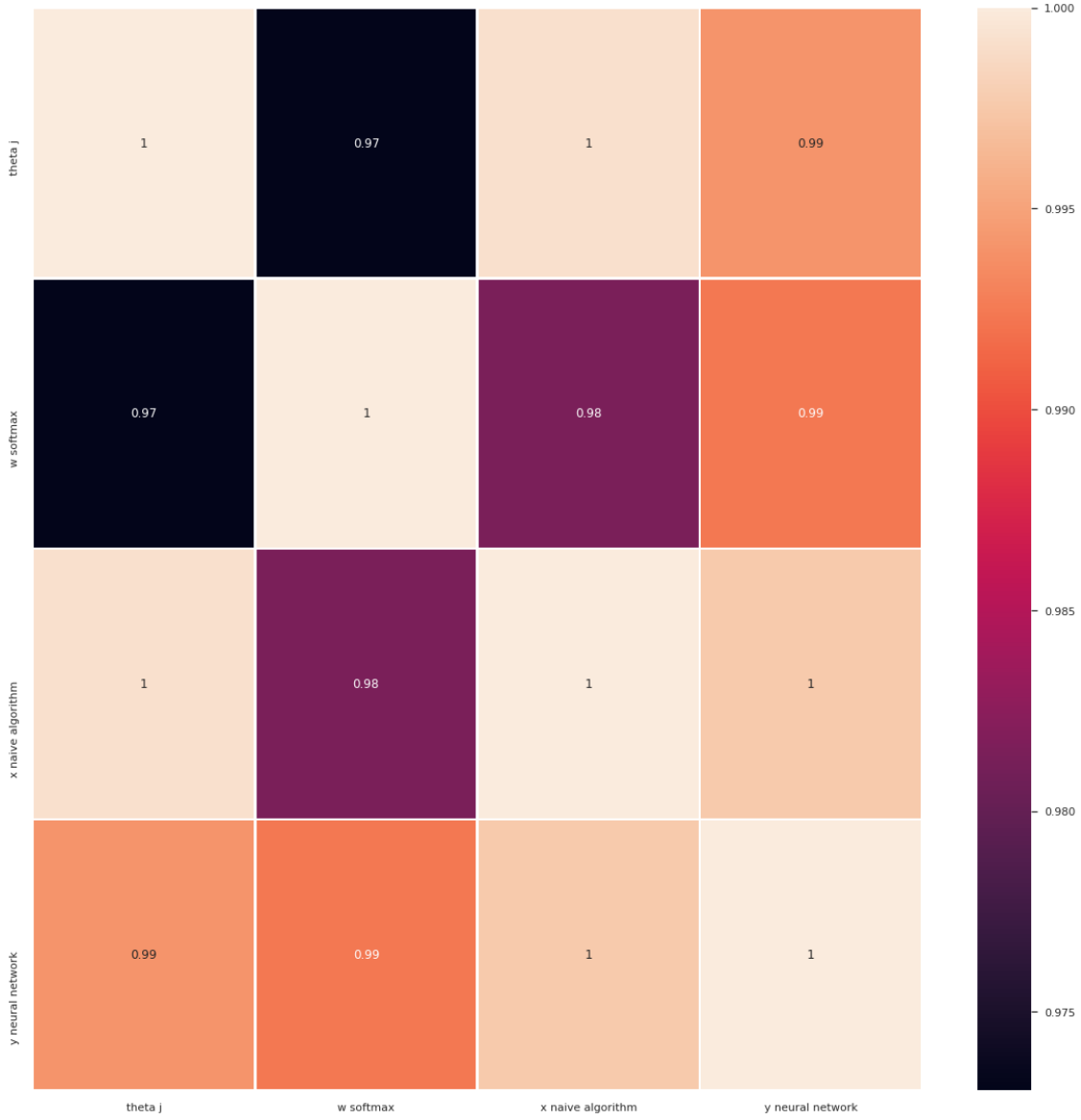
Comparison between the dependent algorithm

[100]:
```
plt.figure(figsize=(15,20))
plt.plot(w_regret_total.mean(0),c='red',    label='w softmax')
plt.plot(x_regret_total.mean(0),c='blue',   label='x naive algorithm')
plt.plot(y_regret_total.mean(0),c='green',  label='y neural network ')


plt.title("Regret Report")
plt.legend(prop={'size': 10})
plt.show()
```



[101]:
```
# correaltion
import pandas as pd
import seaborn as sns; sns.set()
data = pd.DataFrame({
```

```
        'theta j':gt_prob,
        'w softmax':b_estimation[1],
        'x naive algorithm':c_estimation[1],
        'y neural network':d_estimation[1],
    })
plt.figure(figsize=(20,20))
sns.heatmap(data.corr(),annot=True,linewidths=1.5)
plt.show()
```



REFERENCE

1. 13 solutions to multi arm bandit problem for non-mathematicians

2. adaptive step size for policy gradient methods
3. Online and Adaptive Machine Learning
4. Multi-armed bandit problems with dependent arms
5. Gradient Bandit Algorithm baseline
6. RL multi armed bandit

Problem 9

Problem Prove gradient bandit lemma $L_t = \Sigma_{a\in A} E[N_t(a)]\Delta_a$
Proof First, let k be the time, and r_k be the reward of that time. We have that $E[E[r_k|a_k]] = E[r_k]$.

Then, we define the Action $A_t = \Sigma_{k=1}^t E[r_k|a_k]$. Because the sum of possibility of $a_k = a$ for all k is 1. So, $E(A_t) = E[\Sigma_{k=1}^t r_k] = E[\Sigma_{a\in A}\Sigma_{k=1}^t r_k\{the\ possibility\ of\ a_k = a\}] = t$.

After that, the total regret has $L_t = E[\Sigma_{k=1}^t(V^* - E[r_k|a_k])] = tV^* - E[\Sigma_{k=1}^t E[r_k|a_k]] = tV^* - E(A_t) = \Sigma_{a\in A}\Sigma_{k=1}^t E[(V^* - r_k)\{the\ possibility\ of\ a_k = a\}]$.

Because we have $E[(V^* - r_k)\{the\ possibility\ of\ a_k = a\}|a_k] = \{the\ possibility\ of\ a_k = a\}E[(V^* - r_k)|a_k] = \{the\ possibility\ of\ a_k = a\}(V^* - E[r_k|a_k]) = \{the\ possibility\ of\ a_k = a\}\Delta_a = E[\{the\ possibility\ of\ a_k = a\}\Delta_a]$

Eventually, we have $L_t = \Sigma_{a\in A}\Sigma_{k=1}^t E[\{the\ possibility\ of\ a_k = a\}\Delta_a] = \Sigma_{a\in A} E[\Sigma_{k=1}^t\{the\ possibility\ of\ a_k = a\}]\Delta_a = \Sigma_{a\in A} E[N_t(a)]\Delta_a$

Problem 10

Proof First, we should figure out whats the real goal of gradient bandit. That is to maximize the funtion $E[R_t]$ by stochastic acent algorithm, where we apply the update function $H_{t+1}(a) \leftarrow H_t(a) + \alpha\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}$, this is actually stochasitic apprximation to gradient ascent.

The measure of performance here is the expected reward $\mathbb{E}[R_t] = \Sigma_x \pi_t(x) q_*(x)$

We have if $x = a$, $\frac{\partial e^{H(x)}}{\partial H(a)} = e^{H(x)}$, and when $x \neq a$, $\frac{\partial e^{H(x)}}{\partial H(a)} = 0$. Also $\frac{\partial \Sigma_{y=1}^k e^{H(y)}}{\partial H(a)} = e^{H(a)}$

Also, we have $\frac{\partial \pi(x)}{\partial H(a)} = \frac{\partial \frac{e^{H(x)}}{\Sigma_{y=1}^k e^{H(y)}}}{\partial H(a)} = \frac{\frac{\partial e^{H(x)}}{\partial H(a)} * \Sigma_{y=1}^k e^{H(y)} - \frac{\partial \Sigma_{y=1}^k e^{H(y)}}{\partial H(a)} * e^{H(x)}}{(\Sigma_{y=1}^k e^{H(y)})^2}$.

If $x = a$, we have $\frac{\partial \pi(x)}{\partial H(a)} = \frac{\frac{\partial e^{H(x)}}{\partial H(a)} * \Sigma_{y=1}^k e^{H(y)} - \frac{\partial \Sigma_{y=1}^k e^{H(y)}}{\partial H(a)} * e^{H(x)}}{(\Sigma_{y=1}^k e^{H(y)})^2} = \frac{e^{H(x)}}{\Sigma_{y=1}^k e^{H(y)}} - \frac{e^{H(a)}}{\Sigma_{y=1}^k e^{H(y)}} * \frac{e^{H(x)}}{\Sigma_{y=1}^k e^{H(y)}} = \pi(x)(1 - \pi(a))$.

If $x \neq a$, we have $\frac{\partial \pi(x)}{\partial H(a)} = \frac{\frac{\partial e^{H(x)}}{\partial H(a)} * \Sigma_{y=1}^k e^{H(y)} - \frac{\partial \Sigma_{y=1}^k e^{H(y)}}{\partial H(a)} * e^{H(x)}}{(\Sigma_{y=1}^k e^{H(y)})^2} = \frac{e^{H(a)}}{\Sigma_{y=1}^k e^{H(y)}} * \frac{e^{H(x)}}{\Sigma_{y=1}^k e^{H(y)}} = \pi(x)\pi(a)$.

Besides, the exact performance gradient can be calculated by $\frac{\partial E[R_t]}{\partial H_t(a)} = \frac{\partial}{\partial H_t(a)}[\Sigma_x \pi_t(x) q_*(x)] = \Sigma_x \frac{\partial \pi_t(x)}{\partial H_t(a)} q_*(x) = \Sigma_x \frac{\partial \pi_t(x)}{\partial H_t(a)}(q_*(x) - B_t)$, where B_t is called baseline. It can be any scalar that does not depend on x. We can include a baseline here without changing the equality because the gradient sums to

Problem 10 continued on next page...

zero over all the actions $\Sigma \frac{\partial \pi_t(x)}{H_t(a)} = 0$. Thus the sum of the changes must be zero because the sum of the probabilities is always one.

Next we multiply each term of the sum by $\pi_t(x)/\pi_t(x)$: $\frac{\partial E[R_t]}{\partial H_t(a)} = \Sigma_x \pi_t(x)(q_*(x) - B_t)\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x) = E[(q_*(A_t) - B_t)\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x)] = E[(R_t - \bar{R_t})\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x)]$, where A_t is the random variable and notice that $E[R_t|A_t] = q_*(A_t)$.

Take in the $\frac{\partial \pi(x)}{\partial H(a)}$, we have: $E[(R_t - \bar{R_t})\frac{\partial \pi_t(x)}{\partial H_t(a)}/\pi_t(x)] = E[(R_t - \bar{R_t}(1_{a=A_t} - \pi_t(a)))]$

Overall, we have: $H_{t+1}(j) \leftarrow H_t(j) + \alpha(R_t - \bar{R_t})(1_{a=A_t} - \pi_t(j))$