BOUJNOUNI_Fatine_TP2

November 26, 2018

1 Assignement 2 : The Exploration-Exploitation Dilemma

1.1 1 - Stochastic Multi-Armed Bandits on Simulated Data

1.1.1 1.1 Bernoulli bandit models

In [1]:
```
# Imports
import numpy as np
import arms
from tqdm import tqdm
import matplotlib.pyplot as plt
import random
```

In [2]:
```
# Defining our own Bernoulli bandit model with 4 arms

# Random state
rs = np.random.randint(1, 312414)

# Bernouilli Bandit
arm1 = arms.ArmBernoulli(0.50, random_state=rs)
arm2 = arms.ArmBernoulli(0.35, random_state=rs)
arm3 = arms.ArmBernoulli(0.40, random_state=rs)
arm4 = arms.ArmBernoulli(0.25, random_state=rs)

arm5 = arms.ArmBernoulli(0.45, random_state=rs)
arm6 = arms.ArmBernoulli(0.65, random_state=rs)
arm7 = arms.ArmBernoulli(0.15, random_state=rs)
arm8 = arms.ArmBernoulli(0.85, random_state=rs)

MAB1 = [arm1, arm2, arm3, arm4]
MAB2 = [arm5, arm6, arm7, arm8]
```

In [3]:
```
def UCB1(T, MAB, rho=0.2 ):
    """
    Simulates a bandit game of length T with the UCB1 strategy on the bandit model MAB
    Returns "rew": the sequence of the T rewards obtained
        and "draws":  the sequence the T the arms drawn.
```

```
    Parameters :
    ==========
        T : int , Rounds
        MAB : list , List of arms
        ro : float,
    """
    # nbrA : number of arms
    nbrA = len(MAB)
    # List of the obtained rewards
    rew = []
    # List of drawn arms
    draws = []
    # Sum of arms rewards
    sum_rew = [0] * nbrA
    # Number of times each arm has been drawn
    n_draws = [0] * nbrA

    # Initialise first phase : Play each arm once
    for i in range(nbrA):
        n_draws[i] += 1
        reward = int(MAB[i].sample())
        sum_rew[i] += reward
        draws.append(i)
        rew.append(reward)

    # Other drawings until time T
    for t in range(nbrA, T):
        # optimistic scores of the arms at time t
        optimistic_scores = np.array([sum_rew[a]/n_draws[a] + rho*np.sqrt(np.log(t)/(2
                                       for a in range(nbrA)])

        # Pull arm
        # Arm to draw is the arm with the highest score
        index_arm_draw = np.argmax(optimistic_scores)

        reward = int(MAB[index_arm_draw].sample())
        n_draws[index_arm_draw] += 1
        sum_rew[index_arm_draw] += reward
        draws.append(index_arm_draw)
        rew.append(reward)
    return rew, draws
```

```
In [4]: def TS(T,MAB):
            """
            Simulates a bandit game of length T with the Thompson Sampling strategy on the ban
            Returns "rew": the sequence of the T rewards obtained
                and "draws": the sequence the T the arms drawn.
```

```
    Parameters :
    ==========
        T : int, Rounds
        MAB : list, List of arms
    """
    # nbrA : number of arms
    nbrA = len(MAB)
    # List of the obtained rewards
    rew = []
    # List of drawn arms
    draws = []
    # Sum of arms rewards
    sum_rew = [0] * nbrA
    # Number of times each arm has been drawn
    n_draws = [0] * nbrA

    for t in range(T):
        # posterior distributions
        scores = [np.random.beta(sum_rew[a] + 1, n_draws[a] - sum_rew[a] + 1)
                  for a in range(nbrA)]
        # Pull arm
        # Arm to draw is the arm with the highest score
        index_arm_draw = np.argmax(scores)

        reward = int(MAB[index_arm_draw].sample())

        n_draws[index_arm_draw] += 1
        sum_rew[index_arm_draw] += reward
        draws.append(index_arm_draw)
        rew.append(reward)
    return rew, draws
```

```
In [5]: def NaiveStrat(T, MAB):
    """
    Simulates a bandit game of length T with the Naive strategy on the bandit model MA
    Returns "rew": the sequence of the T rewards obtained
        and "draws": the sequence the T the arms drawn.

    Parameters :
    ==========
        T : int, Rounds
        MAB : list, List of arms
    """
    # nbrA : number of arms
    nbrA = len(MAB)
    # List of the obtained rewards
    rew = []
    # List of drawn arms
```

```
        draws = []
        # Sum of arms rewards
        sum_rew = [0] * nbrA
        # Number of times each arm has been drawn
        n_draws = [0] * nbrA

        # Initialise first phase : Play each arm once
        for i in range(nbrA):
            reward = int(MAB[i].sample())
            n_draws[i] += 1
            sum_rew[i] += reward
            draws.append(i)
            rew.append(reward)

        # Other drawings until time T
        for t in range(nbrA, T):
            # Empirical best arm
            scores = np.array([sum_rew[a]/n_draws[a] for a in range(nbrA)])
            # Pull arm
            # Arm to draw is the arm with the highest score
            index_arm_draw = np.argmax(scores)

            reward = int(MAB[index_arm_draw].sample())
            n_draws[index_arm_draw] += 1
            sum_rew[index_arm_draw] += reward
            draws.append(index_arm_draw)
            rew.append(reward)
        return rew, draws
```

In [6]:
```
"""Simulating a bandit game of length T with the UCB1 and Thompson Sampling
strategy on the bandit model MAB: rew and draws are the sequence of the
T rewards obtained and of the T the arms drawn."""
T = 5000  # horizon

rew1, draws1 = UCB1(T, MAB1)
rew2, draws2 = TS(T, MAB1)
rew3, draws3 = NaiveStrat(T, MAB1)
```

In [7]:
```
def expected_regret(MAB, T, strategy, N):
    """Based on many simulations on the MAB for a given strategy,
    it computes the mean regrets at each time.
    It returns an array of mean regrets at each t in range(T)

    Parameters :
    ===========
    MAB : list, list of arms
    T : int, Time horizon
    strategy : str, "UCB1" or "TS" ( Thompson Sampling)
```

```
        N : int, number of simulations
        """
        # best arm
        means = [el.mean for el in MAB]
        mu_max = np.max(means)

        reg = np.zeros((N, T))
        rew = np.zeros(T)
        draws = np.zeros(T)

        for k in tqdm(range(N), desc="Simulating {}".format(strategy)):
            if strategy == "UCB1":
                rew1, draws1 = UCB1(T, MAB)
                reg[k, :] = mu_max * np.arange(1, T + 1) - np.cumsum(rew1)

            elif strategy == "TS":
                rew2, draws2 = TS(T, MAB)
                reg[k, :] = mu_max * np.arange(1, T + 1) - np.cumsum(rew2)

            elif strategy == "NaiveStrat":
                rew3, draws3 == NaiveStrat(T, MAB)
                reg[k, :] =  mu_max * np.arange(1, T + 1) - np.cumsum(rew3)

        mean_regret = np.mean(reg, axis = 0)
        return mean_regret
```

```
In [8]: def kl(x, y):
            return x*np.log(x/y) + (1-x)*np.log((1-x)/(1-y))

        def problem_complexity(MAB):
            means = [arm.mean for arm in MAB]
            p1 = max(means)
            c = sum((p1-p)/(kl(p, p1)) for p in means if p != p1)
            return c
```

Question 1:

```
In [9]: """ Based on many simulations, estimate the expected regret of
        UCB1 and Thompson Sampling """
        N = 100  # number of simulations
        # The expected regret of UCB1
        print ("The expected regret of UCB1 after {} simulations : ".format(N))
        reg1 = expected_regret(MAB1, T, "UCB1", N)
        print(reg1)
        # The expected regret of Thompson Sampling
        print ("The expected regret of Thompson Sampling after {} simulations : ".format(N))
        reg2 = expected_regret(MAB1, T, "TS", N)
```

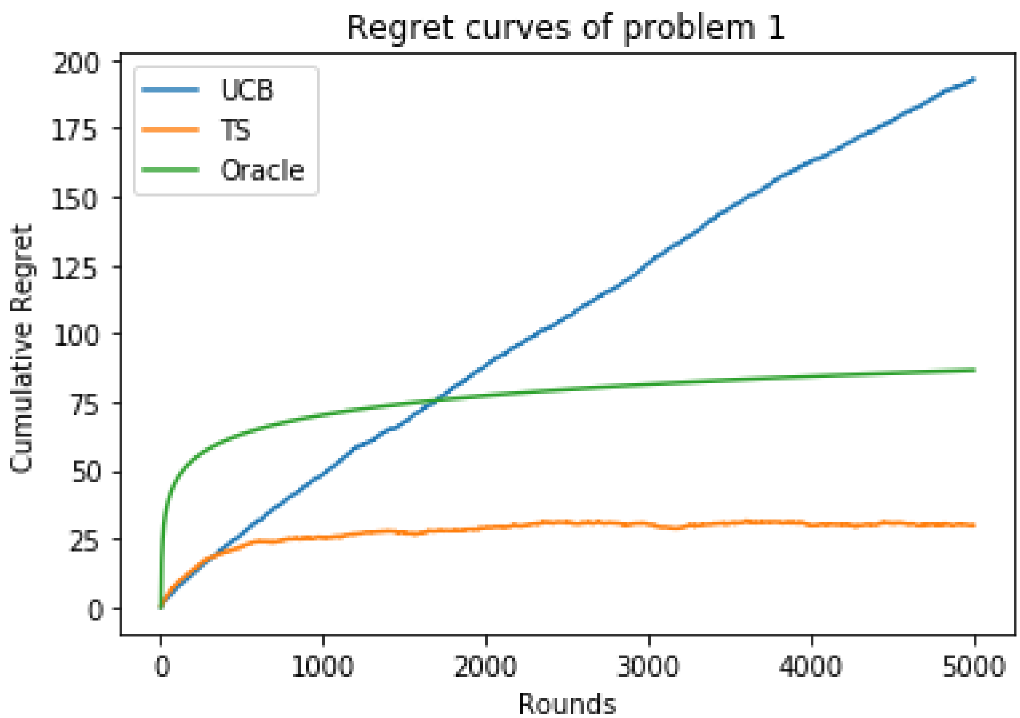
```
print(reg2)
# The expected regret of Naive strategy
print ("The expected regret of Naive strateg after {} simulations : ".format(N))
reg3 = expected_regret(MAB1, T, "NaiveStrat", N)
print(reg3)
```

```
Simulating UCB1:   1%|          | 1/100 [00:00<00:14,  6.86it/s]

The expected regret of UCB1 after 100 simulations :


Simulating UCB1: 100%|| 100/100 [00:16<00:00,  5.51it/s]
Simulating TS:   1%|          | 1/100 [00:00<00:10,  9.25it/s]

[-5.0000e-02  4.0000e-02  1.2000e-01 ...  1.9266e+02  1.9270e+02
  1.9270e+02]
The expected regret of Thompson Sampling after 100 simulations :


Simulating TS: 100%|| 100/100 [00:09<00:00, 10.98it/s]
Simulating NaiveStrat:   2%|         | 2/100 [00:00<00:06, 15.64it/s]

[ 0.26  0.45  0.61 ... 30.08 30.06 29.95]
The expected regret of Naive strateg after 100 simulations :


Simulating NaiveStrat: 100%|| 100/100 [00:05<00:00, 18.16it/s]

[  0.5   1.    0.5 ... 465.  464.5 465. ]
```

In [10]:
```
"""Display regret curves for problem 1"""

x = np.arange(1, T+1)
c = problem_complexity(MAB1)
oracle = [c*np.log(t) for t in x]

print("Problem 1 with complexity :", c)
plt.figure(1)


plt.plot(x, reg1, label='UCB')
plt.plot(x, reg2, label='TS')
#plt.plot(x, reg3, label='Naive')
plt.plot(x, oracle, label='Oracle')
plt.legend()
```

```
plt.xlabel('Rounds')
plt.ylabel('Cumulative Regret')
plt.title('Regret curves of problem 1')
plt.show()
```

```
Problem 1 with complexity : 10.159725560246056
```



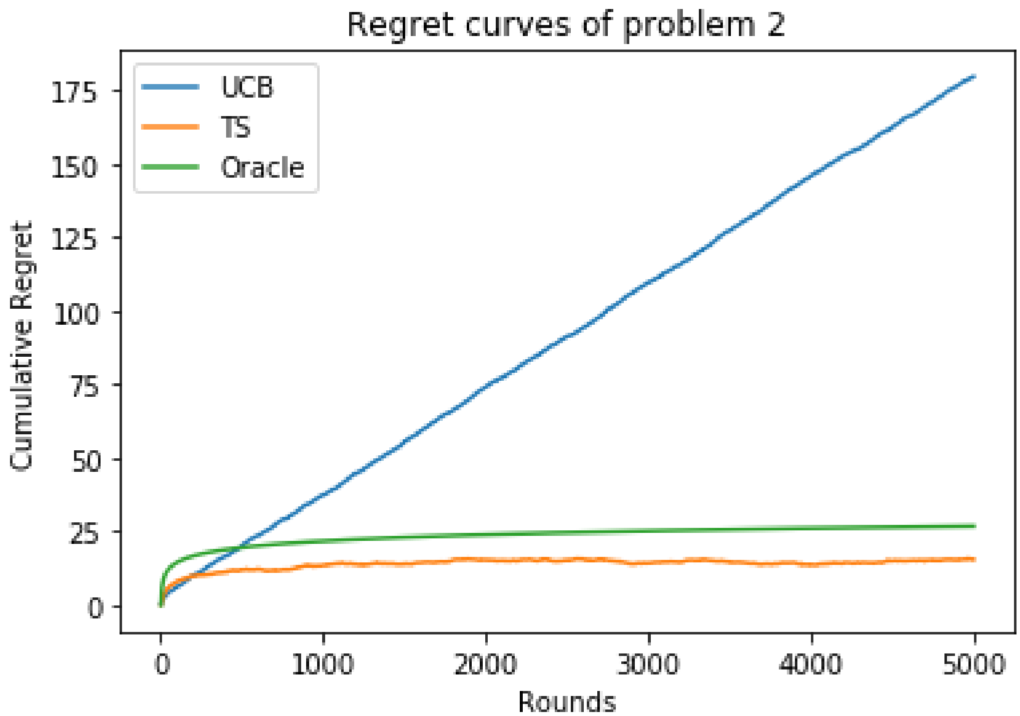

```
In [11]: """ Based on many simulations, estimate the expected regret of
         UCB1 and Thompson Sampling """
         # The expected regret of UCB1
         print ("The expected regret of UCB1 after {} simulations : ".format(N))
         reg1 = expected_regret(MAB2, T, "UCB1", N)
         print(reg1)
         # The expected regret of Thompson Sampling
         print ("The expected regret of Thompson Sampling after {} simulations : ".format(N))
         reg2 = expected_regret(MAB2, T, "TS", N)
         print(reg2)
         # The expected regret of Naive strategy
         print ("The expected regret of Naive strateg after {} simulations : ".format(N))
         reg3 = expected_regret(MAB2, T, "NaiveStrat", N)
         print(reg3)
```

```
Simulating UCB1:   0%|          | 0/100 [00:00<?, ?it/s]
```

```
The expected regret of UCB1 after 100 simulations :


Simulating UCB1: 100%|| 100/100 [00:15<00:00,  6.82it/s]
Simulating TS:   2%|         | 2/100 [00:00<00:08, 11.19it/s]

[  0.34   0.43   1.14 ... 179.63 179.64 179.64]
The expected regret of Thompson Sampling after 100 simulations :


Simulating TS: 100%|| 100/100 [00:10<00:00,  9.49it/s]
Simulating NaiveStrat:   2%|         | 2/100 [00:00<00:06, 14.41it/s]

[ 0.19  0.47  0.74 ... 15.5  15.49 15.44]
The expected regret of Naive strateg after 100 simulations :


Simulating NaiveStrat: 100%|| 100/100 [00:05<00:00, 17.57it/s]

[8.50000e-01 1.70000e+00 1.55000e+00 ... 2.21430e+03 2.21415e+03
 2.21500e+03]
```

In [12]:
```python
"""Display regret curves for problem 2"""

c = problem_complexity(MAB2)
oracle = [c*np.log(t) for t in x]

print("Problem 2 with complexity :", c)
plt.figure(1)
x = np.arange(1, T+1)

plt.plot(x, reg1, label='UCB')
plt.plot(x, reg2, label='TS')
#plt.plot(x, reg3.cumsum(), label='Naive')
plt.plot(x, oracle, label='Oracle')
plt.legend()
plt.xlabel('Rounds')
plt.ylabel('Cumulative Regret')
plt.title('Regret curves of problem 2')
plt.show()
```

```
Problem 2 with complexity : 3.147078426264685
```



The figures shows that the Cumulative regret for the Oracle and Thompson Sampling are greater in the problem with higher complexity. On the more complex problem, the cumulative regret of UCB1 becomes greater than the lower bound when T > 1500 . Otherwise, on the second problem, the cumulative regret of UCB1 becomes greater than the lower bound earlier (T > 500). On the two problems, the cumulative regret of the Thompson sampling strategy is lower than the lower bound.

1.1.2 1.2 Non-parametric bandits (bounded rewards)

Question 2 : The method of Thompson sampling isn't very suited to arms that take continuous values. It returns the sampled reward of the arm if it is a Bernoulli arm. To make it works on the other arms that aren't Bernoulli, we draw a reward from a Bernoulli distribution with the sampled reward as a parameter

The notion of complexity doesn't make sense anymore, because KL involvec in the computation of the complexity is only calculated for Bernouilli variables.

```
In [13]: def TS2(T,MAB):
             """
             Simulates a bandit game of length T with the Thompson Sampling strategy on the ba
             Returns "rew": the sequence of the T rewards obtained
                 and "draws": the sequence the T the arms drawn.

             Parameters :
             ==========
                 T : int, Rounds
```

```
    MAB : list, List of arms
"""
# nbrA : number of arms
nbrA = len(MAB)
# List of the obtained rewards
rew = []
# List of drawn arms
draws = []
# Sum of arms rewards
sum_rew = [0] * nbrA
# Number of times each arm has been drawn
n_draws = [0] * nbrA

for t in range(T):
    # posterior distributions
    scores = [np.random.beta(sum_rew[a] + 1, n_draws[a] - sum_rew[a] + 1)
              for a in range(nbrA)]
    # Pull arm
    # Arm to draw is the arm with the highest score
    index_arm_draw = np.argmax(scores)
    # Bernoulli trial
    r = MAB[index_arm_draw].sample()
    b = arms.ArmBernoulli(r)
    reward = int(b.sample())

    n_draws[index_arm_draw] += 1
    sum_rew[index_arm_draw] += reward
    draws.append(index_arm_draw)
    rew.append(reward)
return rew, draws
```

```
In [14]: def expected_regret_TS2(MAB, T, N):
             """Based on many simulations on the MAB for strategy TS2,
             it computes the mean regrets at each time.
             It returns an array of mean regrets at each t in range(T)

             Parameters :
             ===========
             MAB : list, list of arms
             T : int, Time horizon
             N : int, number of simulations
             """
             # best arm
             means = [el.mean for el in MAB]
             mu_max = np.max(means)


             reg = np.zeros((N, T))
```

```
        rew = np.zeros(T)
        draws = np.zeros(T)

        for k in tqdm(range(N), desc="Simulating TS2"):

            rew2, draws2 = TS2(T, MAB)
            reg[k, :] = mu_max * np.arange(1, T + 1) - np.cumsum(rew2)

        mean_regret = np.mean(reg, axis = 0)
        return mean_regret
```

In [15]:
```
# Non-parametric bandits

# Random state
rs = np.random.randint(1, 312414)

arm1 = arms.ArmBernoulli(0.50, random_state=rs)
arm2 = arms.ArmBeta(0.35, 0.2, random_state=rs)
arm3 = arms.ArmExp(L=1, random_state=rs)
arm4 = arms.ArmExp(L=1.5, random_state=rs)

MAB = [arm1, arm2, arm3, arm4]
```

In [16]:
```
N = 100  # number of simulations
# The expected regret of UCB1
print ("The expected regret of UCB1 after {} simulations : ".format(N))
reg1 = expected_regret(MAB, T, "UCB1", N)
print(reg1)
# The expected regret of Thompson Sampling
print ("The expected regret of Thompson Sampling after {} simulations : ".format(N))
reg2 = expected_regret_TS2(MAB, T, N)
print(reg2)
# The expected regret of Naive strategy
print ("The expected regret of Naive strateg after {} simulations : ".format(N))
reg3 = expected_regret(MAB, T, "NaiveStrat", N)
print(reg3)
```

```
Simulating UCB1:   0%|          | 0/100 [00:00<?, ?it/s]

The expected regret of UCB1 after 100 simulations :


Simulating UCB1: 100%|| 100/100 [00:15<00:00,  6.80it/s]
Simulating TS2:   1%|          | 1/100 [00:00<00:16,  6.00it/s]

[1.46363636e-01 7.82727273e-01 1.41909091e+00 ... 6.82275455e+02
 6.82421818e+02 6.82608182e+02]
The expected regret of Thompson Sampling after 100 simulations :
```

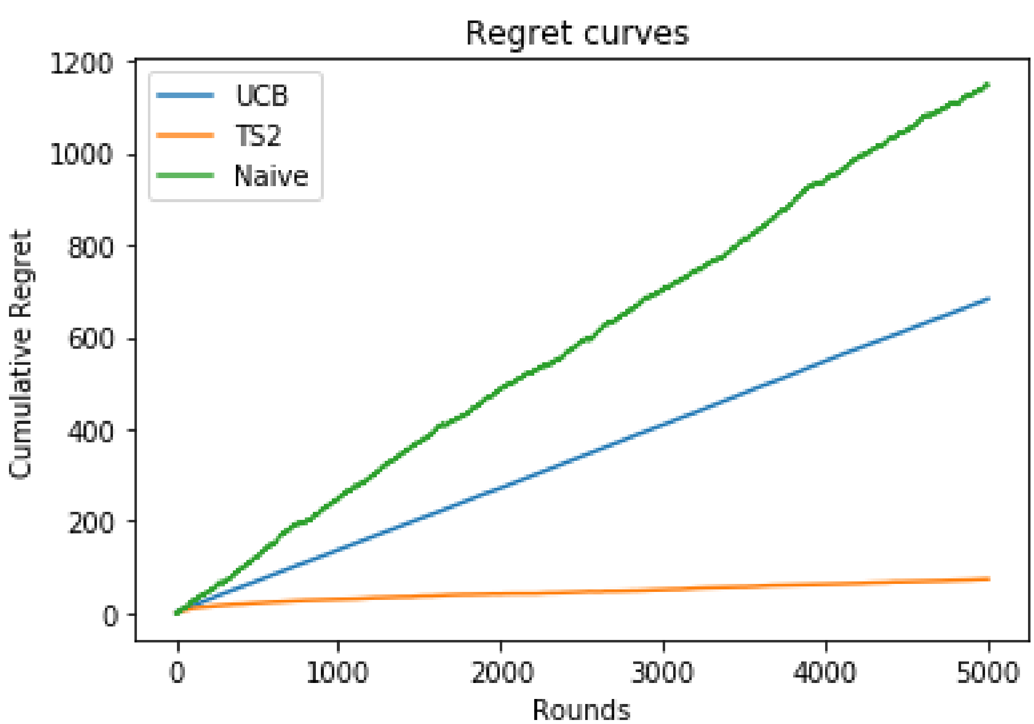
```
Simulating TS2: 100%|| 100/100 [00:16<00:00,  6.42it/s]
Simulating NaiveStrat:   2%|          | 2/100 [00:00<00:05, 18.67it/s]
```

```
[ 0.19636364  0.48272727  0.67909091 ... 73.94545455 73.93181818
 73.94818182]
The expected regret of Naive strateg after 100 simulations :
```

```
Simulating NaiveStrat: 100%|| 100/100 [00:05<00:00, 19.94it/s]
```

```
[6.36363636e-01 1.27272727e+00 9.09090909e-01 ... 1.14654545e+03
 1.14618182e+03 1.14681818e+03]
```

In [17]:
```
"""Display regret curves"""

plt.figure(1)
x = np.arange(1, T+1)
plt.plot(x, reg1, label='UCB')
plt.plot(x, reg2, label='TS2')
plt.plot(x, reg3, label='Naive')
plt.legend()
plt.xlabel('Rounds')
plt.ylabel('Cumulative Regret')
plt.title('Regret curves ')
plt.show()
```



The naive strategy has the greatest cumulative regret, and the Thompson strategy has the smallest cumulative regret.

1.2 2 - Linear Bandit on Real Data

1.2.1 2.1 - Linear Bandit

1.2.2 Question 3:

```
In [18]: def linUCB(model, T, nb_simu, alpha, lamb):
             """ Linear Bandit Problem
             It returns the mean regrets and the mean norms of the model
             Parameters :
             ============
             model : class defined in linearmab_models.py
             T : int , Time horizon
             nb_simu : int, nombre de simulation
             alpha : float
             lamb : float
             """
             nbrA = model.n_actions
             nbrF = model.n_features

             regret = np.zeros((nb_simu, T))
             norm_dist = np.zeros((nb_simu, T))

             F = model.features.T
             I = np.identity(nbrF)

             for k in tqdm(range(nb_simu), desc="Simulating LinUCB"):

                 Z = np.zeros((0,nbrF))
                 y = np.zeros((0))

                 for t in range(T):

                     if len(Z) == 0 and len(y) == 0:
                         X = np.identity(nbrF)
                         Y = np.zeros(nbrF)
                     else:
                         X = np.dot(Z.T,Z) + lamb*I
                         Y = np.dot(Z.T,y)


                     theta_hat = np.dot(np.linalg.inv(X),Y)

                     A = np.dot(F.T,theta_hat)
```

```
                beta = alpha*np.sqrt(np.diag(np.dot(np.dot(F.T,np.linalg.inv(X)), F)))

                a_t = np.argmax(A + beta)  # pick action
                r_t = model.reward(a_t) # get the reward

                Z = np.append(Z, [F[:,a_t].T], axis = 0)
                y = np.append(y, r_t, axis = 0)

                regret[k, t] = model.best_arm_reward() - r_t
                norm_dist[k, t] = np.linalg.norm(theta_hat - model.real_theta, 2)

        mean_norms = np.mean(norm_dist,axis = 0)
        mean_regret = np.mean(regret,axis = 0)

        return mean_norms, mean_regret
```

```
In [619]: def linUCB(model, T, nb_simu, alpha, lamb):
              """ Linear Bandit Problem
              It returns the mean regrets and the mean norms of the model
              Parameters :
              ============
              model : class defined in linearmab_models.py
              T : int , Time horizon
              nb_simu : int, nombre de simulation
              alpha : float
              lamb : float
              """
              regret = np.zeros((nb_simu, T)) # regret at each iteration
              norm_dist = np.zeros((nb_simu, T)) #
              nbrF = model.n_features # number of features
              nbrA = model.n_actions # number of actions
              A = np.zeros(nbrA)

              for k in tqdm(range(nb_simu), desc="Simulating LinUCB "):

                  # Initialization
                  X = lamb*np.identity(nbrF)   # Z_t.T*Z_t + lambda*Id
                  Y = np.zeros(nbrF)  #Z_t.T*y_t

                  for t in range(T):
                      theta_hat = np.dot(np.linalg.inv(X),Y).reshape(-1,1)

                      for a in range(nbrA):
                          A[a] = np.dot(model.features[a,:], theta_hat) + alpha*np.sqrt(model.

                      a_t = np.argmax(A )

                      r_t = model.reward(a_t) # get the reward
```

```python
            F = model.features[a_t, :].reshape(-1,1)
            X += F.dot(F.T)
            Y += r_t*F.flatten()

            # store regret
            regret[k, t] = model.best_arm_reward() - r_t
            norm_dist[k, t] = np.linalg.norm(theta_hat - model.real_theta, 2)

    # compute average (over sim) of the algorithm performance and plot it
    mean_norms =  np.mean(norm_dist, axis = 0)
    mean_regrets = np.mean(regret, axis = 0)

    return mean_regrets, mean_norms
```

```python
In [19]: def Random_policy(model, T, nb_simu, lamb):
            """   """
            regret = np.zeros((nb_simu, T)) # regret at each iteration
            norm_dist = np.zeros((nb_simu, T)) #
            nbrF = model.n_features # number of features

            for k in tqdm(range(nb_simu), desc="Simulating Random policy "):

                # Initialization
                X = lamb*np.identity(nbrF)   # Z_t.T*Z_t + lambda*Id
                Y = np.zeros(nbrF)   #Z_t.T*y_t


                for t in range(T):
                    theta_hat = np.dot(np.linalg.inv(X),Y)

                    a_t = np.random.randint(model.n_actions)   # random arm
                    r_t = model.reward(a_t) # get the reward

                    F = model.features[a_t, :].reshape(-1,1)
                    X += F.dot(F.T)
                    Y += r_t*F.flatten()

                    # store regret
                    regret[k, t] = model.best_arm_reward() - r_t
                    norm_dist[k, t] = np.linalg.norm(theta_hat - model.real_theta, 2)

            # compute average (over sim) of the algorithm performance and plot it
            mean_norms =  np.mean(norm_dist, axis = 0)
            mean_regrets = np.mean(regret, axis = 0)

            return mean_regrets, mean_norms
```

In [20]:
```python
def epsilon_greedy(model, T, nb_simu, epsilon, lamb):
    """   """
    regret = np.zeros((nb_simu, T)) # regret at each iteration
    norm_dist = np.zeros((nb_simu, T)) #
    nbrF = model.n_features # number of features

    for k in tqdm(range(nb_simu), desc="Simulating $\epsilon$-greedy "):

        # Initialization
        X = lamb*np.identity(nbrF)   # Z_t.T*Z_t + lambda*Id
        Y = np.zeros(nbrF)  #Z_t.T*y_t


        for t in range(T):
            theta_hat = np.dot(np.linalg.inv(X),Y)

            # chooses a random arm with probability epsilon and the optimal
            # arm with probability 1-epsilon
            r = random.random()
            if r < epsilon:
                a_t = np.random.randint(model.n_actions)
            else:
                a_t = np.argmax(np.dot(model.features, theta_hat))

            r_t = model.reward(a_t) # get the reward

            F = model.features[a_t, :].reshape(-1,1)
            X += F.dot(F.T)
            Y += r_t*F.flatten()

            # store regret
            regret[k, t] = model.best_arm_reward() - r_t
            norm_dist[k, t] = np.linalg.norm(theta_hat - model.real_theta, 2)

    # compute average (over sim) of the algorithm performance and plot it
    mean_norms =  np.mean(norm_dist, axis = 0)
    mean_regrets = np.mean(regret, axis = 0)

    return mean_regrets, mean_norms
```

In [21]:
```python
from linearmab_models import ToyLinearModel, ColdStartMovieLensModel

random_state = np.random.randint(0, 24532523)

model = ColdStartMovieLensModel(random_state=random_state,noise=0.1)
```

In [22]:
```python
T = 6000
```

```
nb_simu = 50
alpha = 50
lamb = 1
epsilon = 0.1
```

In [23]:
```
mean_regrets1, mean_norm1 = linUCB(model, T, nb_simu,alpha, lamb)
```

```
Simulating LinUCB: 100%|| 50/50 [03:29<00:00,  4.00s/it]
```

In [24]:
```
print(mean_norm1)
```

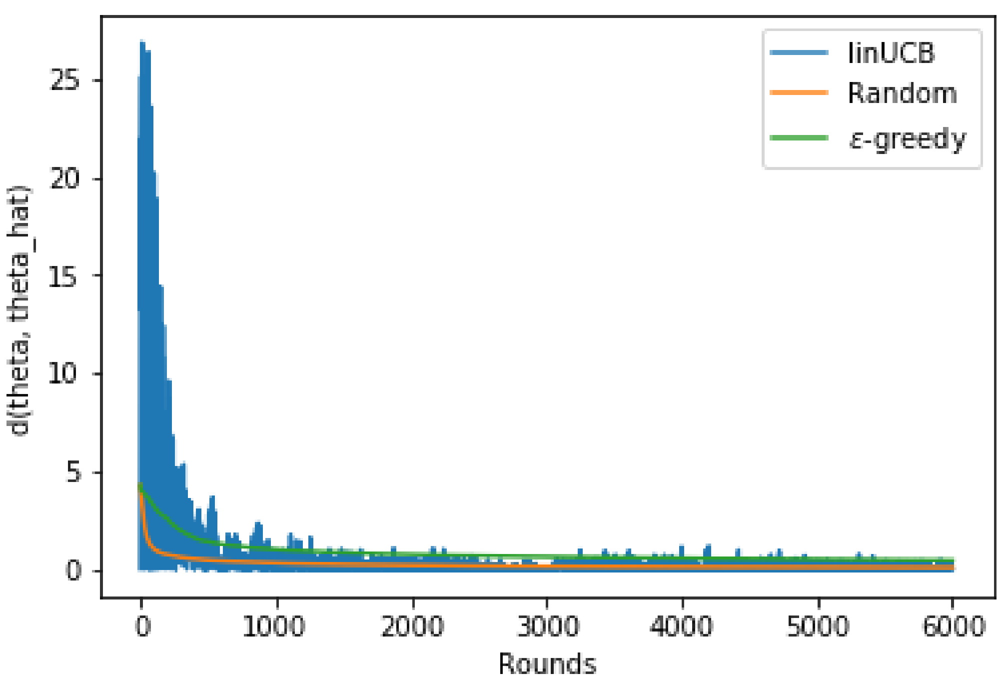
```
[-2.02064895e-03  2.15704212e+01  2.19280492e+01 ...  3.26602799e-02
  1.38894100e-02 -5.77988575e-03]
```

In [25]:
```
mean_regrets2, mean_norm2 = Random_policy(model, T, nb_simu, lamb)
mean_regrets3, mean_norm3 = epsilon_greedy(model, T, nb_simu, epsilon, lamb)
```

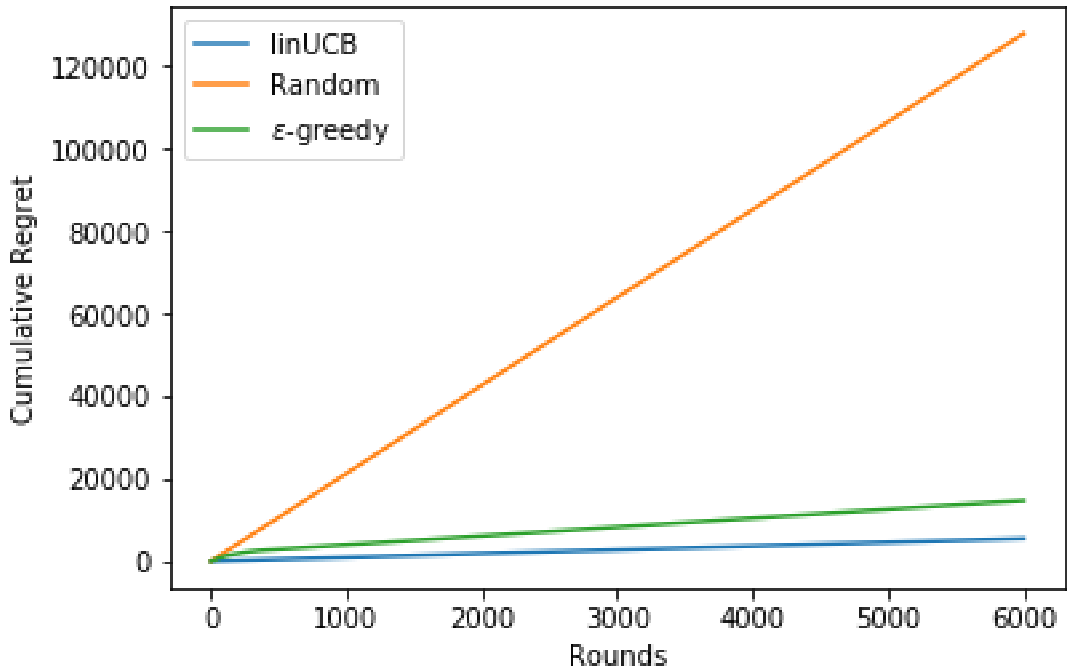
```
Simulating Random policy : 100%|| 50/50 [00:29<00:00,  1.73it/s]
Simulating $\epsilon$-greedy : 100%|| 50/50 [00:33<00:00,  1.53it/s]
```

In [26]:
```
plt.plot(mean_norm1, label='linUCB')
plt.plot(mean_norm2, label='Random')
plt.plot(mean_norm3, label='$\epsilon$-greedy')
plt.ylabel('d(theta, theta_hat)')
plt.xlabel('Rounds')
plt.legend()
```

Out[26]: <matplotlib.legend.Legend at 0x1131df908>



```
In [27]: plt.plot(mean_regrets1.cumsum(), label='linUCB')
         plt.plot(mean_regrets2.cumsum(), label='Random')
         plt.plot(mean_regrets3.cumsum(), label='$\epsilon$-greedy')
         plt.ylabel('Cumulative Regret')
         plt.xlabel('Rounds')
         plt.legend()
         plt.show()
```



The random strategy converges faster than the other strategies. The linUCB algorithm converges to the real θ but it's more noisy.

The linUCB has the lower cumulative regrets. And the random strategy has the highest cumulative regret.

```
In [ ]:
```