OREISTEIN_Pierre_TP2

November 26, 2018

1 1 - Information

```
In [1]: author = "Pierre OREISTEIN"
        date = "25/11/2018"
```

2 2 - Stochastic Multi-Armed Bandits on Simulated Data

2.1 2.1 - Packages

```
In [2]: # Mathematical packages
        import numpy as np

        # Graphic packages
        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn as sns
        sns.set()

        # Progress bar
        import tqdm as tqdm

        # Personnal packages
        import arms as arms
```

2.2 2.2 - UCB1

```
In [3]: def UCB1(T, MAB, n=1, rho=0.2):
            """Simulate a bandit game of length T on the MAB given as argument with the UCB1
               strategy."""

            # Sequence of T rewards and of the T arms drawn
            rew = []
            draws = []

            # Number of arms
            K = len(MAB)
```

```python
# Initialisation of the approxiamted means, counter and sum of each arms
sums = np.zeros(K)
N_a = np.zeros(K)
means = np.zeros(K)

# Initialisation of the time
t = 0

# First: Draw each arm
while t < (T - 1) and t < K:

    # Arm to draw
    k = t

    # Draw a sample of arm k
    sample_k = MAB[k].sample()

    # Update the sum, the mean and the counter of arm k
    sums[k] += sample_k
    N_a[k] += 1
    means[k] += sample_k

    # Update the time
    t += 1

    # Update rew and draws
    rew.append(sample_k)
    draws.append(k)

# Second: Chooses the best arm in the best possible world
while t < (T - 1):

    # Compute the upper bounds
    A = means + rho * np.sqrt(np.log(t) / (2 * N_a))

    # Compute the argmax
    k_max = np.argmax(A)

    # Draw a sample of arm k
    sample_k_max = MAB[k_max].sample()

    # Update the sum, the mean and the counter of arm k
    sums[k_max] += sample_k_max
    N_a[k_max] += 1
    means[k_max] = sums[k_max] / N_a[k_max]

    # Update the time
    t += 1
```

```
            # Update rew and draws
            rew.append(sample_k_max)
            draws.append(k_max)

        return [rew, draws]
```

2.3 2.3 - Thomson Sampling

```
In [4]: class UniformArm(object):
            def __init__(self):
                """
                Args:
                    random_state (int): seed to make experiments reproducible
                """
                self.local_random = np.random.RandomState(np.random.randint(1, 312414))

            def sample(self):
                return self.local_random.uniform()
```

```
In [5]: def TS(T, MAB):
            """Simulate a bandit game of length T on the MAB given as argument
               with the Thomson Sampling strategy."""

            # Sequence of T rewards and of the T arms drawn
            rew = []
            draws = []

            # Number of arms
            K = len(MAB)

            # Initialisation of the approxiamted distributions
            distributions = np.array([UniformArm() for i in range(K)])

            # Initialisation of the approxiamted sums, means and counter of each arms
            sums = np.zeros(K)
            N_a = np.zeros(K)
            means = np.zeros(K)

            # Initialisation of the time
            t = 0

            # Chooses the best arm according to the sample from the approximated laws
            while t < (T - 1):

                # Compute the distributions
                for k in range(K):
```

```
            # Extract S_a_t and N_a_t
            N_a_t = N_a[k]
            S_a_t = sums[k]

            # Compute the new distribution
            distributions[k] = arms.ArmBeta(S_a_t + 1, N_a_t - S_a_t + 1)

        # Extract a sample for each law
        samples = [arm.sample() for arm in distributions]

        # Compute the argmax
        k_max = np.argmax(samples)

        # Draw a sample of arm k
        sample_k_max = MAB[k_max].sample()

        # Update the sum, the mean and the counter of arm k
        sums[k_max] += sample_k_max
        N_a[k_max] += 1
        means[k_max] = sums[k_max] / N_a[k_max]

        # Update the time
        t += 1

        # Update rew and draws
        rew.append(sample_k_max)
        draws.append(k_max)

    return [rew, draws]
```

2.4 2.4 - Naive Strategy

```
In [6]: def naiveStrategy(T, MAB):
            """Select the empirical best arm at each round."""

            # Sequence of T rewards and of the T arms drawn
            rew = []
            draws = []

            # Number of arms
            K = len(MAB)

            # Initialisation of the approxiamted sums, means and counter of each arms
            sums = np.zeros(K)
            means = np.zeros(K)
            N_a = np.zeros(K)

            # Initialisation of the time
```

```python
t = 0

# First: Draw each arm
while t < (T - 1) and t < K:

    # Arm to draw
    k = t

    # Draw a sample of arm k
    sample_k = MAB[k].sample()

    # Update the mean and the counter of arm k
    sums[k] += sample_k
    means[k] += sample_k
    N_a[k] += 1

    # Update the time
    t += 1

    # Update rew and draws
    rew.append(sample_k)
    draws.append(k)

# Second: Chooses the best arm in the best possible world
while t < (T - 1):

    # Compute the argmax
    k_max = np.argmax(means)

    # Draw a sample of arm k
    sample_k_max = MAB[k_max].sample()

    # Update the sum, the mean and the counter of arm k
    sums[k_max] += sample_k_max
    N_a[k_max] += 1
    means[k_max] = sums[k_max] / N_a[k_max]

    # Update the time
    t += 1

    # Update rew and draws
    rew.append(sample_k_max)
    draws.append(k_max)

return [rew, draws]
```

2.5 2.5 - Estimation of the expected regret with many simulations

In [7]:
```
def regretEstimation(T, MAB, n=500, strategy=UCB1):
    """This function estimates the expected regret trough n simulations."""

    # Extract the maximal mean
    means = [el.mean for el in MAB]
    p_star = np.max(means)

    # Initialisation of the mean reward
    reward = 0

    # Simulate n parties
    for i in range(n):

        # Result of the i-th simulation
        rew, draws = strategy(T, MAB)

        # Update the reward
        reward += np.array(rew).sum()

    # Mean of the rewards
    reward /= n

    return T * p_star - reward
```

2.6 2.6 - Computation of the values of the oracle

In [8]:
```
def KL(p_1, p_2):
    """Compute the Kullback-Liebler divergence for two different Bernoulli laws."""

    result = p_1 * np.log(p_1 / p_2) + (1 - p_1) * np.log((1 - p_1) / (1 - p_2))

    return result
```

In [9]:
```
def complexity(MAB):
    """Compute the complexity of our problem."""

    # Extract the probabilities of the Bernoulli laws.
    p = np.array([el.mean for el in MAB])
    p_star = np.max(p)

    # Initialisation of the complexity
    C = 0

    # Compute the complexity
    for a in range(len(p)):

        # Extract the current p
```

```
            p_a = p[a]

            if p_a != p_star:

                # Compute the divergence
                kl_a = KL(p_a, p_star)

                # Update the complexity
                C += (p_star - p_a) / kl_a

        return C
```

```
In [10]: def oracleCurve(MAB, t):
             """Compute the value of the oracle according to the model given as argument and t

             # Compute the complexity
             C = complexity(MAB)

             return C * np.log(t)
```

2.7 2.7 - Plotting functions

```
In [11]: def plotRegretCurves(MAB, nb_T=5, max_T=2500, n_MC=500, oracle=True):
             """Plot the Regret curves over the number of iterations for MAB."""

             # Parameters of the figure
             plt.figure(figsize=(12,8))
             plt.grid(True)

             # Array of the different T
             T_l = np.linspace(1, max_T, nb_T, dtype=int)

             # Array of the strategies
             strategies = [UCB1, TS, naiveStrategy]

             # Plot the regret curve for the different strategies
             for s in tqdm.tqdm(range(len(strategies))):

                 # Compute the regret for each T for the given strategy
                 regret_l = []
                 for t in range(len(T_l)):
                     regret_l.append(regretEstimation(T_l[t], MAB, n=n_MC, strategy=strategies

                 # Plot
                 plt.scatter(T_l, regret_l, label=strategies[s].__name__, marker="x")

             # Add the oracle curve
             if oracle:
```

```
            voracleCurve = np.vectorize(lambda t: oracleCurve(MAB, t))
            oracle_l = voracleCurve(T_l)
            plt.scatter(T_l, oracle_l, label="Oracle", marker="x")

            # Save the plot
            name = str(round(complexity(MAB), 2))
            plt.savefig("./Images/Regret_Curves_C_=_" + name + ".eps", bbox_inches='tight

        # Legend
        plt.xlabel("Rounds")
        plt.ylabel("Cumulative Regret")
        plt.legend()

        # Display
        plt.show()
```

2.8 2.8 - Computation of the regret for different strategies

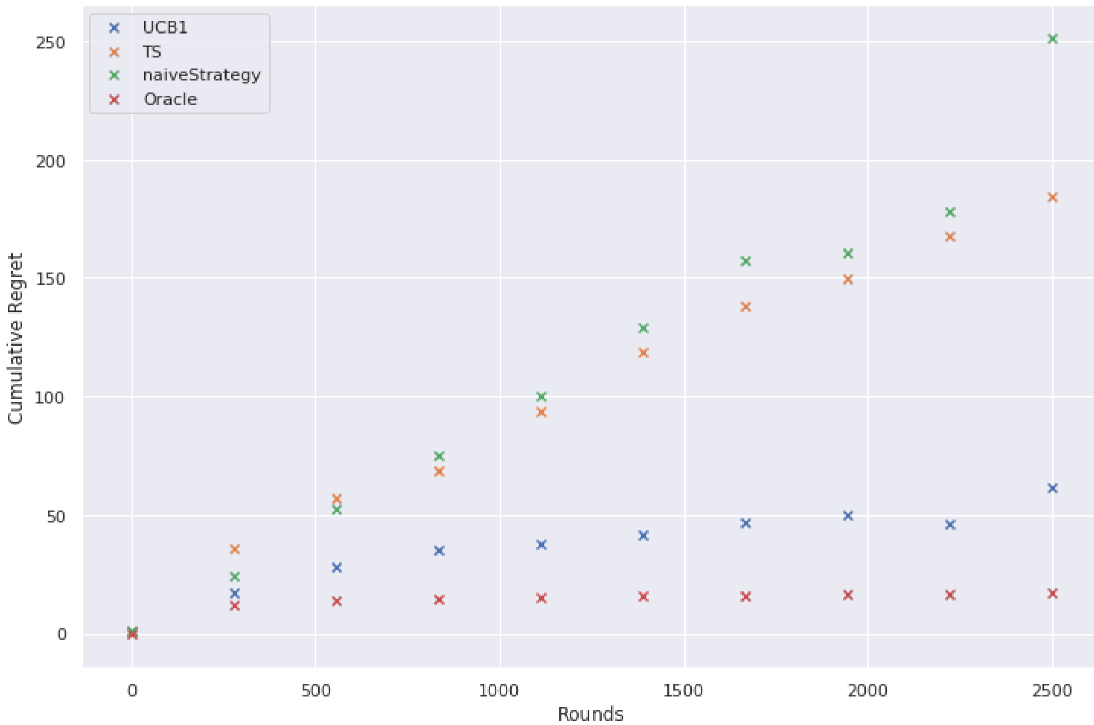
2.8.1 2.8.1 - Low Complexity: C = 2.16

```
In [12]: # Definition of Multi-Bandit Arms
         arm1_1 = arms.ArmBernoulli(0.25, random_state=np.random.randint(1, 312414))
         arm2_1 = arms.ArmBernoulli(0.30, random_state=np.random.randint(1, 312414))
         arm3_1 = arms.ArmBernoulli(0.35, random_state=np.random.randint(1, 312414))
         arm4_1 = arms.ArmBernoulli(0.85, random_state=np.random.randint(1, 312414))
         MAB_1 = [arm1_1, arm2_1, arm3_1, arm4_1]

         # Display the complexity of this model
         print(complexity(MAB_1))
```

```
2.1620936547523386
```

```
In [25]: # Displaying of the regret curves
         plotRegretCurves(MAB_1, nb_T=10, max_T=2500, n_MC=1000)
```

```
100%|| 3/3 [17:37<00:00, 282.87s/it]
```



- First, we can observe that the naive strategy is the worst. In fact, if we look at it more closely, we can understand that the value of the regret will depend a lot on the initialisation of the means of the different arms. In particular, if the sample of initialisation of the best arm is equal to 0 (here we use Bernoulli law), it will be very unlikely to use that arm later.
- Second, we can observe that the Thomson-Sampling strategies perform a little bit better than the naive one. In fact, the best arm of our MAB is pretty easy to detect. So, even if the exploitation of the arms is better managed by the TS algorithm, it is not so advantageous in comparison with the naive strategy because at the same time it is harder for the TS algorithm to learn the true distribution trough Beta distribution.
- Third, we can observe that neither the Thomson-Sampling strategy nor the UCB1 strategy perform better than the lower bound.
- Fourth, the UCB1 performs the best. It seems logic. In fact, as the sub-optimal arms are really less efficient than the best one, the exploration is really simplified and so we can expect that the UCB1 algorithm to detect really quickly the best arm and so to draw it much more often than for the TS strategy which need more iterations.

2.8.2 2.8.2 - Model with high complexity: C = 18

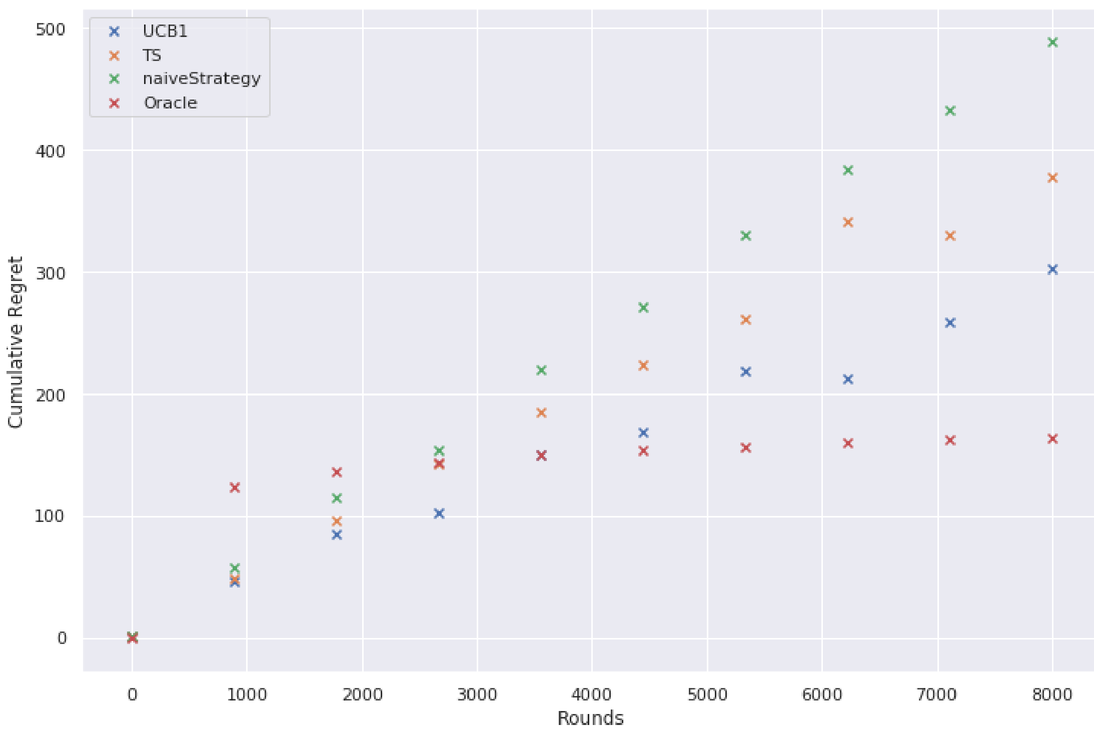
```
In [13]: # Definition of Multi-Bandit Arms
         arm1_2 = arms.ArmBernoulli(0.35, random_state=np.random.randint(1, 312414))
         arm2_2 = arms.ArmBernoulli(0.40, random_state=np.random.randint(1, 312414))
```

```
arm3_2 = arms.ArmBernoulli(0.45, random_state=np.random.randint(1, 312414))
arm4_2 = arms.ArmBernoulli(0.50, random_state=np.random.randint(1, 312414))
MAB_2 = [arm1_2, arm2_2, arm3_2, arm4_2]

# Display the complexity of this model
print(complexity(MAB_2))
```

```
18.231880696697445
```

```
In [14]: # Displaying of the regret curves
         plotRegretCurves(MAB_2, nb_T=10, max_T=8000, n_MC=400)
```

```
100%|| 3/3 [20:37<00:00, 331.99s/it]
```



- Here, we can observe that the scale of the cumulated regret is bigger than the one in the past model, in particular for the lower bound. In fact, in this new model, the arms have probabilities very close. Hence it is hard to learn which arm is the best and so the regret increase quickly.
- Second, we can observe that the strategies UCB1 and Thomson-Sampling continue to increase rapidly, even after 7000 iterations. It is caused by the high complexity of our model. In fact, as the arms have very close probability, the number of times a sub-optimal arm will be drawn is going to be much higher than before.

- Third, as before we can observe that the cumulated regret of all the strategies becomes bigger than the lower bound after some iterations. However, we can observe that at the beginning, all the strategies performs better than the lower bound. In fact, the lower bound is true only at the infinite, so it still coherent.

2.9 2.9 - Non parametric Bandit Games

2.9.1 2.9.1 - Adaptation of the Thomson Sampling Algorithm

Here we adapt the Thomson Sampling Algorithm to manage arms with non binary output. For doing that, we use the work done by *Shipra Agrawal and Navin Goyal*, in *Analysis of thompson sampling for the multi-armed bandit problem.* that can be found at http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf.

The modifications are quite simple if we consider only laws with a support inside [0,1]. At each round, we still continue to draw the arm that maximises:

$$\theta_a^t, \text{ where } \theta_a^t \sim \quad _a(t) := Beta(S_a(t)+1; N_a(t)S_a(t)+1) \tag{1}$$

Let's call a^* the maximising arm.

Then, we continue to draw the arm a^* and we obtain a reward $\tilde{r}_t$. Next, instead of using directly that reward as before, we use it as the parameter of a Bernoulli law $\mathcal{B}(\tilde{r}_t)$. We execute this Bernoulli law and we obtain a new binary reward r_t. Finally, we use it as the as the normal reward and the next computations does not change.

```
In [14]: def TS2(T, MAB):
             """Simulate a bandit game of length T on the MAB given as argument
                with the Thomson Sampling strategy adapted for non Bernoulli law."""

             # Sequence of T rewards and of the T arms drawn
             rew = []
             draws = []

             # Number of arms
             K = len(MAB)

             # Initialisation of the approxiamted distributions
             distributions = np.array([UniformArm() for i in range(K)])

             # Initialisation of the approxiamted sums, means and counter of each arms
             sums = np.zeros(K)
             N_a = np.zeros(K)
             means = np.zeros(K)

             # Initialisation of the time
             t = 0

             # Chooses the best arm according to the sample from the approximated laws
             while t < (T - 1):
```

```python
        # Compute the distributions
        for k in range(K):

            # Extract S_a_t and N_a_t
            N_a_t = N_a[k]
            S_a_t = sums[k]

            # Compute the new distribution
            distributions[k] = arms.ArmBeta(S_a_t + 1, N_a_t - S_a_t + 1)

        # Extract a sample for each law
        samples = [arm.sample() for arm in distributions]

        # Compute the argmax
        k_max = np.argmax(samples)

        # Draw a sample of arm k
        sample_k_max = MAB[k_max].sample()

        # Peform a Bernoulli test with porbability sample_k_max
        # http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf
        r_t_delta = np.random.rand()
        if r_t_delta < sample_k_max:
            r_t = 1
        else:
            r_t = 0

        # Update the sum, the mean and the counter of arm k
        sums[k_max] += r_t
        N_a[k_max] += 1
        means[k_max] = sums[k_max] / N_a[k_max]

        # Update the time
        t += 1

        # Update rew and draws
        rew.append(sample_k_max)
        draws.append(k_max)

    return [rew, draws]
```

2.9.2 2.9.2 - The lower bound holds also for more general probability distributions

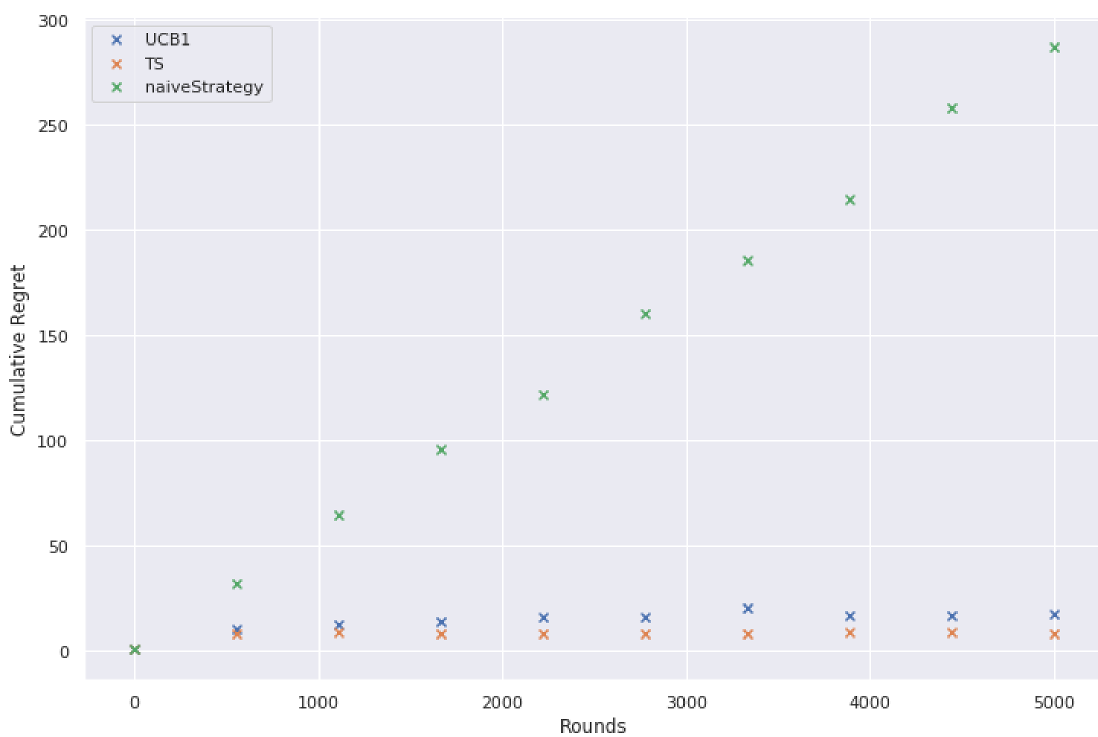
According to *Burnetas and Katehakis* and this blog "blogs.princeton.edu/imabandit/2016/05/11/bandit-theory-part-i/" , the lower bound holds for more general probability distribution and the complexity notion can be adapted for more genral probability distribution. Moreover, we can remark that in the past algorithm all happen as if the mean of the different laws corresponded to the probability p of the Bernoulli law. So, at the end the complexity still make sense.

2.9.3 2.9.3 - Definition of a new MAB

In [15]:
```
# Definition of Multi-Bandit Arms
arm1_3 = arms.ArmBernoulli(0.35, random_state=np.random.randint(1, 312414))
arm2_3 = arms.ArmBeta(a=5, b=10, random_state=np.random.randint(1, 312414))
arm3_3 = arms.ArmExp(L=2, random_state=np.random.randint(1, 312414))
# For the arm finite, we have to take care to consider a support inside [0,1]
arm4_3 = arms.ArmFinite(X=np.array([0.25, 0.15, 0.75, 0.5]),
                        P=np.array([0.2, 0.3, 0.4, 0.1]),
                        random_state=np.random.randint(1, 312414))
MAB_3 = [arm1_3, arm2_3, arm3_3, arm4_3]

# Display the complexity of this model
# print(complexity(MAB_3))
```

In [29]:
```
# Displaying of the regret curves
plotRegretCurves(MAB_3, nb_T=10, max_T=2500, n_MC=1000, oracle=False)
```

```
100%|| 3/3 [1:18:57<00:00, 1432.26s/it]
```



3 3 - Linear Bandit on Real Data

3.1 3.1 - Packages

```
In [16]: from linearmab_models import ToyLinearModel, ColdStartMovieLensModel
```

3.2 3.2 - LinUCB

```
In [17]: def beta_t(alpha, inv, phi, K):
             """Compute beta_t for all actions."""

             # Parameters
             K, d = np.shape(phi)

             # Initialisation of the saving beta
             beta_t = []

             # Compute beta_t_a for all action
             for a in range(K):

                 # Extract phi_a_t
                 phi_a_t = phi[a, :]

                 # Compute beta_t_a
                 beta_t_a = alpha * np.sqrt(np.dot(phi_a_t.T, np.dot(inv, phi_a_t)))

                 # Compute beta_t_a
                 beta_t.append(beta_t_a)

             # Reshape the array of beta_t
             beta_t = np.array(beta_t).reshape((-1, 1))

             return beta_t

In [18]: def A_t_recursive(phi_a_t, A_t_1):
             """Return the value of A_t for the given model."""

             # Reshape phi_a_t
             phi_a_t = phi_a_t.reshape((-1, 1))

             # Compute A_t
             A_t = A_t_1 + np.dot(phi_a_t, phi_a_t.T)

             return A_t

In [19]: def b_t_recursive(phi_a_t, r_a_t, b_t_1):
             """Compute the next iteration of b_t."""

             # Reshape phi_a_t
```

```python
        phi_a_t = phi_a_t.reshape((-1, 1))

        # Compute the next iteration of b
        b_t = b_t_1 + phi_a_t * r_a_t

        return b_t
```

```python
In [54]: def LinUCB(model, T, alpha=1, lambda_const=1):
             """Executes the Linear UCB for the model given as argument until T."""

             # Parameters
             K = model.n_actions
             d = model.n_features

             # Sequence of T rewards and of the T arms drawn
             rew = []
             draws = []

             # Extraction of the phi
             phi = model.features

             # Initialisation of theta, A_t, b_t
             theta_t = np.random.rand(d).reshape((d, 1))
             A_t = lambda_const * np.eye(d, d)
             b_t = 0

             # Compute the inverse of A_t
             inv = np.linalg.inv(A_t)

             # Initialisation of the time
             t = 0

              # Second: Chooses the best arm in the best possible world
             while t < (T - 1):

                 # Compute the bound beta_t
                 beta_t_value = beta_t(alpha, inv, phi, K)

                 # Compute the upper bounds
                 R = np.dot(phi, theta_t) + beta_t_value

                 # Compute the argmax
                 a_t = np.argmax(R)

                 # Extract phi_a_t
                 phi_a_t = phi[a_t, :]

                 # Draw a sample of arm a
```

```
        r_a_t = model.reward(a_t)

        # Update the time, A_t and b_t
        t += 1
        A_t = A_t_recursive(phi_a_t, A_t)
        b_t = b_t_recursive(phi_a_t, r_a_t, b_t)

        # Compute the inverse of A_t and theta_t
        inv = np.linalg.inv(A_t)
        theta_t = np.dot(inv, b_t).reshape((-1, 1))

        # Update rew and draws
        rew.append(r_a_t)
        draws.append(phi_a_t)

    return [rew, draws, theta_t]
```

3.3 3.3 - Random policy

```
In [84]: def randomPolicy(model, T, lambda_const=1):
             """Executes the random policy over the model given as argument until T."""

             # Parameters
             K = model.n_actions
             d = model.n_features

             # Sequence of T rewards and of the T arms drawn
             rew = []
             draws = []

             # Extraction of the phi
             phi = model.features

             # Initialisation of the time
             t = 0

             # Initialisation of theta, A_t, b_t
             theta_t = np.random.rand(d).reshape((d, 1))
             A_t = lambda_const * np.eye(d, d)
             b_t = 0

             # Second: Chooses the best arm in the best possible world
             while t < (T - 1):

                 # Choose a random action
                 a_t = np.random.randint(0, K)

                 # Extract phi_a_t
```

```
        phi_a_t = phi[a_t, :]

        # Draw a sample of arm a
        r_a_t = model.reward(a_t)

        # Update the time, A_t and b_t
        t += 1
        A_t = A_t_recursive(phi_a_t, A_t)
        b_t = b_t_recursive(phi_a_t, r_a_t, b_t)

        # Compute the inverse of A_t and theta_t
        inv = np.linalg.inv(A_t)
        theta_t = np.dot(inv, b_t).reshape((-1, 1))

        # Update rew and draws
        rew.append(r_a_t)
        draws.append(phi_a_t)

    return [rew, draws, theta_t]
```

3.4 3.4 - Greedy Policy

```
In [56]: def epsilonGreedy(model, T, epsilon=0.1, lambda_const=1):
    """Executes the epsilon greedy policy over the model given as argument until T."""

    # Parameters
    K = model.n_actions
    d = model.n_features

    # Sequence of T rewards and of the T arms drawn
    rew = []
    draws = []

    # Extraction of the phi
    phi = model.features

    # Initialisation of theta, A_t, b_t
    theta_t = np.random.rand(d).reshape((d, 1))
    A_t = lambda_const * np.eye(d, d)
    b_t = 0

    # Initialisation of the time
    t = 0

     # Second: Chooses the best arm in the best possible world
    while t < (T - 1):

        # Compute the upper bounds
```

```
        R = np.dot(phi, theta_t)

        # Compute the argmax
        a_t = np.argmax(R)

        # Choose the random policy with probability epsilon
        rand = np.random.rand()
        if rand < epsilon:
            # Choose a random action
            a_t = np.random.randint(0, K)

        # Extract phi_a_t
        phi_a_t = phi[a_t, :]

        # Draw a sample of arm a
        r_a_t = model.reward(a_t)

        # Update the time, A_t, b_t, theta_t
        t += 1
        A_t = A_t_recursive(phi_a_t, A_t)
        b_t = b_t_recursive(phi_a_t, r_a_t, b_t)
        theta_t = np.dot(np.linalg.inv(A_t), b_t).reshape((-1, 1))

        # Update rew and draws
        rew.append(r_a_t)
        draws.append(phi_a_t)

    return [rew, draws, theta_t]
```

3.5 3.5 - L^2 norm

```
In [23]: def l2Norm(theta, model):
             """Take an array of theta computed by a strategy over different simulations."""

             # Extract real_theta
             real_theta = model.real_theta
             real_theta = real_theta.reshape((1, -1))

             # Compute the L2 norm
             norm = np.mean(np.sqrt(np.sum((real_theta - theta) ** 2, axis=1)))

             return norm
```

3.6 3.6 - Expected Cumulative Regret

```
In [24]: def cumulativeRegret(model, T, rewards_l):
             """Compute the mean regret computed over different simulations."""
```

```
        # Extract the maximal possible reward
        r_star = model.best_arm_reward()

        # Computation of the mean reward
        reward = np.mean(np.sum(rewards_l, axis=1))

        return T * r_star - reward
```

3.7 3.7 - Estimation

```
In [38]: def estimation(model, T, n_MC=1000, strategy=randomPolicy):
        """Run n_MC simulation of the strategy over the model and save the computed rewar
           and theta."""

        # Initialisation of the array of the rewards and theta
        rewards_l = []
        theta_l = []

        # Simulate n parties
        for i in range(n_MC):

            # Result of the i-th simulation
            rew, draws, theta = strategy(model, T)

            # Update the reward
            rewards_l.append(rew)

            # Update theta
            theta_l.append(list(theta.reshape(-1)))

        # Transform the list as array
        rewards_l = np.array(rewards_l)
        theta_l = np.array(theta_l)

        return [rewards_l, theta_l]
```

3.8 3.8 - Definition of our model

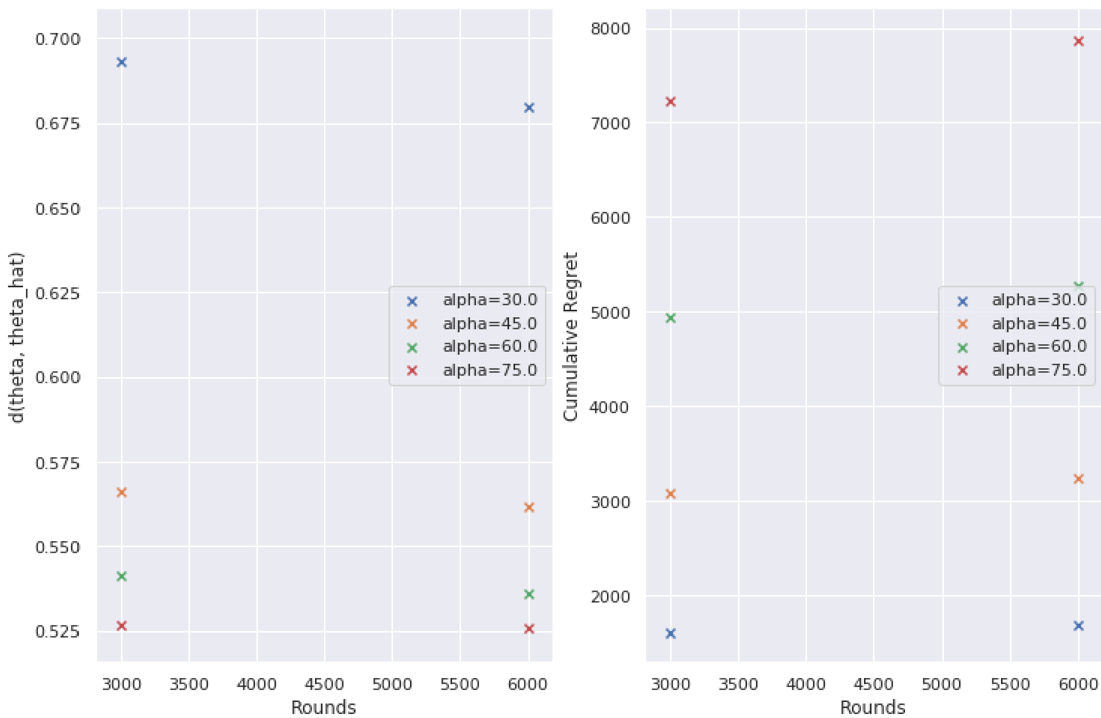
```
In [39]: # Choice of a random state
         random_state = np.random.randint(0, 24532523)

In [40]: # Definition of our model
         model = ColdStartMovieLensModel(
             random_state=random_state,
             noise=0.1
         )
```

3.9 3.9 - Estimation of the best alpha for the LinUCB

In [41]:
```
def gridSearchAlpha(model, min_alpha=30, max_alpha=75, nb_alpha=4,
                    lambda_const=1,
                    nb_T=4, max_T=6000, n_MC=20):
    """Plot the result for the two metrics for the different alpha and lambda."""

    # Parameters of the figure
    fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12,8))
    plt.grid(True)

    # Array of the different T
    T_l = [3000, 6000] #np.linspace(1, max_T, nb_T, dtype=int)

    # Array of the value for alpha
    alpha_l = np.linspace(min_alpha, max_alpha, nb_alpha, dtype=float)

    # Plot the regret curve for the different strategies
    for s in range(len(alpha_l)):

        # Extract alpha
        alpha = round(alpha_l[s], 2)

        # Strategy to test
        strategy_alpha = lambda model, t: LinUCB(model, t, alpha=alpha,
                                                 lambda_const=lambda_const)

        # Compute the regret for each T for the given strategy
        regret_l = []
        theta_l = []

        for t in range(len(T_l)):
            rewards_l, thetas_l = estimation(model, T_l[t], n_MC=n_MC,
                                             strategy=strategy_alpha)

            # Compute the score for the two metrics for this (alpha, T)
            regret_l.append(cumulativeRegret(model, T_l[t], rewards_l))
            theta_l.append(l2Norm(thetas_l, model))

        # Plot
        axs[0].scatter(T_l, theta_l, label="alpha=" + str(alpha), marker="x")

        # Plot
        axs[1].scatter(T_l, regret_l, label="alpha=" + str(alpha), marker="x")

    # Legend
    axs[0].set_xlabel("Rounds")
    axs[0].set_ylabel("d(theta, theta_hat)")
```

```
axs[0].legend()

# Legend
axs[1].set_xlabel("Rounds")
axs[1].set_ylabel("Cumulative Regret")
axs[1].legend()

# Save the plot
plt.savefig("./Images/Alpha_Curves.eps", bbox_inches='tight', pad_inches=0.0)

# Display
plt.show()
```

```
In [42]: gridSearchAlpha(model)
```



As the computation were really slow, we just computed the scores for T=3000 and T=6000. Here there is not an optimal value of alpha for the both criterion. It seems logic because bigger is alpha, more important is the exploration. So, it is easier to approximate the right theta. However, it causes also a bigger regret. Hence, we choose $\alpha = 45$ as optimal value because it seems as a good trade-off between the two criterion.

Here, we just display the result for a given lambda, but it easy to launch the past function for different values of λ and then select the one which minimises the errors. However, the results do not change a lot according to λ. So, we choose $\lambda = 1$.

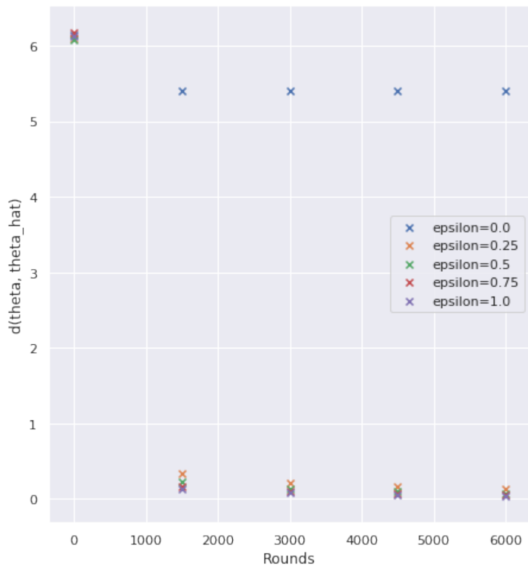
3.10 3.8 - Estimation of the best epsilon

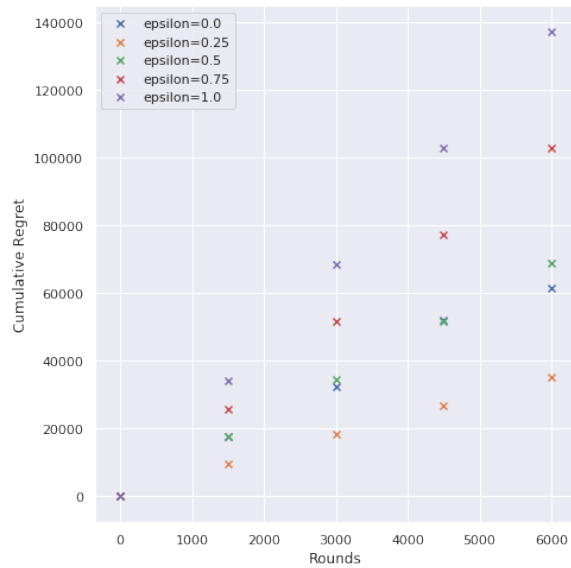
```
In [43]: def gridSearchEpsilon(model, min_epsilon=0, max_epsilon=1, nb_epsilon=5,
                               lambda_const=1,
                               nb_T=5, max_T=6000, n_MC=50):
             """Plot the result for the two metrics for the different alpha."""

             # Parameters of the figure
             fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(16,8))
             plt.grid(True)

             # Array of the different T
             T_l = np.linspace(1, max_T, nb_T, dtype=int)

             # Array of the value for epsilon
             epsilon_l = np.linspace(min_epsilon, max_epsilon, nb_epsilon, dtype=float)

             # Plot the regret curve for the different strategies
             for s in tqdm.tqdm(range(len(epsilon_l))):

                 # Compute the regret for each T for the given strategy
                 regret_l = []
                 theta_l = []

                 # Extract alpha
                 epsilon = round(epsilon_l[s], 2)

                 # Strategy to test
                 strategy_epsilon = lambda model, t: epsilonGreedy(model, t, epsilon=epsilon,
                                                                  lambda_const=lambda_const)

                 for t in range(len(T_l)):

                     rewards_l, thetas_l = estimation(model, T_l[t], n_MC=n_MC,
                                                      strategy=strategy_epsilon)

                     # Compute the score for the two metrics for this (alpha, T)
                     regret_l.append(cumulativeRegret(model, T_l[t], rewards_l))
                     theta_l.append(l2Norm(thetas_l, model))

                 # Plot
                 ax = axs[0]
                 ax.scatter(T_l, theta_l, label="epsilon=" + str(epsilon), marker="x")

                 # Plot
                 ax = axs[1]
                 ax.scatter(T_l, regret_l, label="epsilon=" + str(epsilon), marker="x")
```

```
        # Legend
        axs[0].set_xlabel("Rounds")
        axs[0].set_ylabel("d(theta, theta_hat)")
        axs[0].legend()

        # Legend
        axs[1].set_xlabel("Rounds")
        axs[1].set_ylabel("Cumulative Regret")
        axs[1].legend()

        # Save the plot
        plt.savefig("./Images/Epsilon_Curves.eps", bbox_inches='tight', pad_inches=0.0)

        # Display
        plt.show()
```

```
In [44]: gridSearchEpsilon(model, nb_T=5, n_MC=50, lambda_const=1)
```

```
  0%|          | 0/5 [00:00<?, ?it/s]

 20%|        | 1/5 [02:52<11:31, 172.81s/it]

 40%|      | 2/5 [05:26<08:21, 167.12s/it]

 60%|    | 3/5 [08:07<05:30, 165.38s/it]

 80%|  | 4/5 [10:55<02:45, 165.99s/it]

100%|| 5/5 [15:43<00:00, 202.61s/it]
```





Thanks to these analysis we can deduce that $\epsilon = 0.25$ performs the best. As before, the result doesnot change a lot according to λ. So, we are going to choose $\lambda = 1$.

3.11 3.9 - Plot the result for the different algorithms

According to the past result we select the best parameters for alpha and epsilon.

```
In [85]: # Definition of the best LinUCB and the best epsilonGreedy strategies
         best_LinUCB = lambda model, T: LinUCB(model, T, alpha=45,
                                               lambda_const=1)
         best_LinUCB.__name__ = "LinUCB"
         best_epsilon_Greedy = lambda model, T: epsilonGreedy(model, T, epsilon=0.25,
                                                              lambda_const=1)
         best_epsilon_Greedy.__name__ = "$\epsilon$-Greedy"

         # Definition of the strategies
         strategies = [best_LinUCB,
                       best_epsilon_Greedy,
                       randomPolicy]
```

```
In [50]: def plotMetricsCurves(model, strategies, nb_T=5, max_T=6000, n_MC=20):
             """Plot the result for the two metrics for the different alpha."""

             # Parameters of the figure
             fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(12,8))
             plt.grid(True)

             # Array of the different T
             T_l = np.linspace(1, max_T, nb_T, dtype=int)
```

```
    # Plot the regret curve for the different strategies
    for s in tqdm.tqdm(range(len(strategies))):

        # Compute the regret for each T for the given strategy
        regret_l = []
        theta_l = []

        for t in range(len(T_l)):
            rewards_l, thetas_l = estimation(model, T_l[t], n_MC=n_MC, strategy=strate

            # Compute the score for the two metrics for this (alpha, T)
            regret_l.append(cumulativeRegret(model, T_l[t], rewards_l))
            theta_l.append(l2Norm(thetas_l, model))

        # Plot
        axs[0].scatter(T_l, theta_l, label=strategies[s].__name__, marker="x")

        # Plot
        axs[1].scatter(T_l, regret_l, label=strategies[s].__name__, marker="x")

    # Legend
    axs[0].set_xlabel("Rounds")
    axs[0].set_ylabel("d(theta, theta_hat)")
    axs[0].legend()

    # Legend
    axs[1].set_xlabel("Rounds")
    axs[1].set_ylabel("Cumulative Regret")
    axs[1].legend()

    # Save the plot
    plt.savefig("./Images/Strategies_Curves.eps", bbox_inches='tight', pad_inches=0.0

    # Display
    plt.show()
```

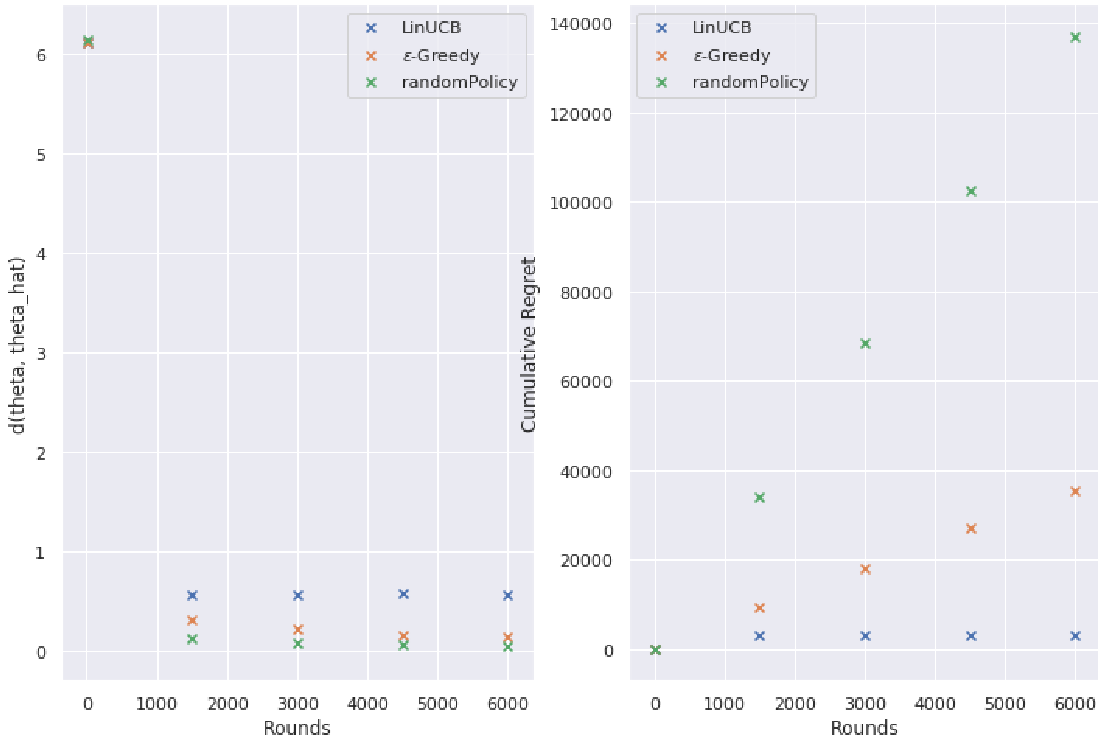
```
In [86]: plotMetricsCurves(model, strategies)
```

```
  0%|          | 0/3 [00:00<?, ?it/s]
```

```
 33%|      | 1/3 [11:23<22:46, 683.19s/it]
```

```
 67%|   | 2/3 [12:24<08:16, 496.52s/it]
```

```
100%|| 3/3 [13:21<00:00, 364.67s/it]
```



- First, we can observe that the randomPolicy performs the best for approximating the right θ. It is not really surprising because it is the policy which explores the best the space of actions. For the same reason, as the ϵ-greedy policy explores the space of actions with probability ϵ, we have good result for the approximation of θ.
- Second, we observe that the LinUCB performs the best for the regret. It seems logic because it is the one which is the best for the exploitation. Also, we can observe that the ϵ-greedy policy continues to increase linearly. It is likely that it is related to the fact that we choose a random action with probability ϵ.

```
In [ ]:
```