Stochastic Linear Bandits An Empirical Study

Students: BARRA Romane, AHYERRE Victoire, **Lecturer:** Claire Vernade

Introduction

Stochastic Linear Bandits are a great way to model decision-making problems where an agent must choose sequentially from a set of actions based on contextual information, while the rewards of these actions are uncertain.

In Linear Bandits, the expected reward for an action is modeled as a linear function of its context, which can be expressed as: $r_t(a) = \theta^T x_t(a) + \alpha$ where $r_t(a)$ is the reward for action a at time t, θ is a vector of unknown parameters, $x_t(a)$ is the feature vector associated with action a at time t, and α is a noise term, assumed to be Gaussian.

The objective of this assignment is to implement and evaluate three key algorithms in the context of Linear Bandits: Linear Epsilon Greedy (LinEpsGreedy), Linear Upper Confidence Bound (LinUCB), and Linear Thompson Sampling (LinTS). These algorithms present three different ways of balancing the trade-off between exploration (trying new actions) and exploitation (choosing the action with the highest expected reward based on past experiences).

1 Problem 1 : Linear Epsilon Greedy

1.1 Action generation

The Action Generator is a function that generates K independent actions in dimension d on the unit sphere. To implement this method, we sampled K times from the isotropic multivariate normal distribution (for independence) and then normalized to have samples in the unit sphere.

```python
def ActionsGenerator(K, d, mean=None):
    if mean is None:
        mean = np.zeros(d)
    actions = np.random.multivariate_normal(mean, np.eye(d), K)
    actions /= np.linalg.norm(actions, axis=1)[:, np.newaxis]
    return actions
```

The following tests are done in environments with random action sets (generated as above).

1.2 LinEpsGreedy Algorithm

Given a set of K actions given by the Action Generator, the Linear Epsilon Greedy algorithm works as follows. With probability ϵ, it randomly chooses an action (*exploration*). Otherwise, it selects the action that maximizes the expected reward based on available information (contextual information and observed rewards : *exploitation*).

The algorithm maintains an estimate of θ based on the data observed up to time t, through the least square estimator. This is done by solving the following equations at each time step:

$$B_t^\lambda = \lambda I_d + \sum_{s=1}^{t} x_s x_s^\top \qquad \hat{\theta}_t^\lambda = (B_t^\lambda)^{-1} \sum_{s=1}^{t} r_s x_s$$

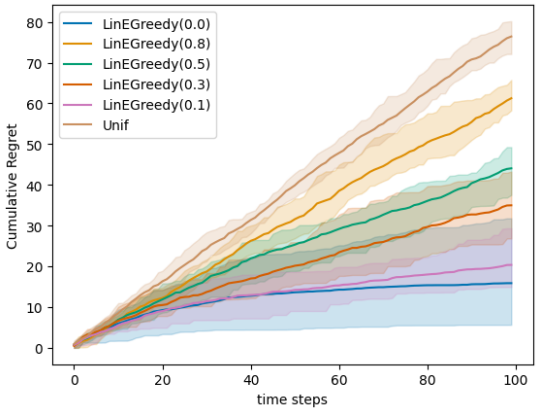


Figure 1: Cumulative Regret of LinEpsGreedy, LinGreedy and LinUniform

The algorithm needs, as parameters : ϵ and the regularization parameter λ.

To evaluate the performance of LinEpsGreedy, we create a simple Linear Bandit environment ($K = 7$ arms of dimension $d = 3$ and a known true vector θ). Then we run LinEpsGreedy and plot the cumulative regret for different values of ϵ ($\lambda = 1$, $N_{mc} = 10$ Monte Carlo experiments, $T = 100$ time steps). We compare LinEpsGreedy with LinUniform and LinGreedy $\epsilon = 0$). The results are presented in Figure 1.

LinUniform's cumulative regret curve is linear whereas for low values of ϵ, LinEpsGreedy performs better with a logarithmic curve : adding exploitation (i.e. decreasing ϵ) decreases the cumulative regret. Here, we observe that the agent with $\epsilon = 0.1$ achieves the best performance, after LinGreedy ($\epsilon = 0$). This suggests that exploitation might be more effective here. Indeed, we chose a basic low dimension problem where the set of actions is random, explaining why exploitation is very convenient. That is why LinGreedy algorithm outperforms every other choice of ϵ in this simple case. This would not be true for more complex problems in higher dimension and with fixed actions (where exploration would be required).

If Linear Epsilon Greedy algorithm can be quite basic and outperformed by more sophisticated algorithms, especially in complex or high-dimensional problems, it remains a reasonable baseline for the Linear Bandit problem.

1.3 Matrix inversion

The main computational cost in the LinEpsGreedy algorithm is the matrix inversion of B_t^λ in the update of θ. The complexity is $O(d^3)$, which is really expensive in high dimension.

To address this issue, we can use an incremental update for B_t^λ, which avoids inverting a matrix at each step. We used the Sherman-Morisson formula :

$$(B_t^\lambda + xx^\top)^{-1} = (B_t^\lambda)^{-1} - \frac{(B_t^\lambda)^{-1}xx^\top(B_t^\lambda)^{-1}}{1 + x^\top(B_t^\lambda)^{-1}x}$$

This update only requires matrix vector multiplications and leads to a complexity of $O(d^2)$.

We ran the following experiment. For different dimensions, we randomly created a Linear Bandit environment with $K = 10$ actions and a vector θ. We ran both LinEpsGreedy with matrix inversion and LinEpsGreedyIncremental with incremental update of $(B_t^\lambda)^{-1}$. For different dimensions d, we performed the same experiment 10 times with either the original or the incremental algorithm. In Figure 2a, the mean relative execution time and the mean relative cumulative regret (performance) are plotted. Figure 2b shows the cumulative regret for the same experience as Figure 1 (dimension $d = 3$) for both methods.

(a) **(b)**

Figure 2: **(a)** Gain in Runtime (%) for LinEpsGreedyIncremental over LinEpsGreedy ($\epsilon = 0.1$), **(b)** Cumulative Regret for LinEpsGreedy and LinEpsGreedyIncremental for different values of ϵ

We observe in 2a that the incremental update approach yields a significant improvement in runtime, especially as the dimension d of the feature increases. LinEpsGreedyIncremental was approximately 3 times faster than LinEpsGreedy for $d = 100$ (relative time of 0.35), while having the same performance.

2 Problem 2 : LinUCB and LinTS

2.1 LinUCB

At each time t, the action is chosen among available actions A_t according to the following procedure : $a_t = argmax_{a \in A_t} a^T \hat{\theta}_t^\lambda + ||a||_{B_t^{\lambda-1}} \beta(t, \delta)$ where $\beta(t, \delta) = \sigma\sqrt{2\log\left(\frac{1}{\delta}\right) + d\log\left(1 + \frac{tL}{d\lambda}\right)} + \sqrt{\lambda}\|\hat{\theta}_t^\lambda\|$ for the chosen δ, σ, L and λ. The idea is to rely on a confidence ellipsoid around an action and choose the best action according to the upper-bound estimation, with an exploration bonus.

2.2 LinTS

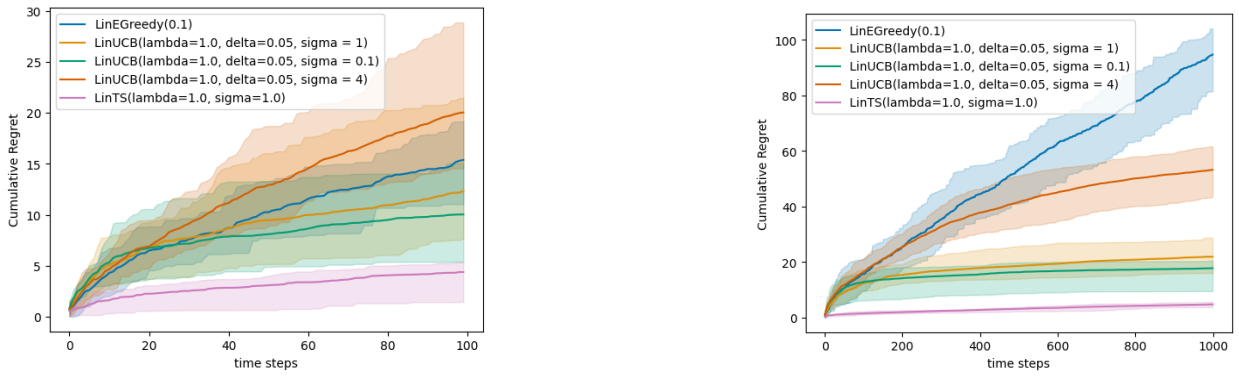
The idea is to approximate θ at each step t by a random sample $\tilde{\theta}$ drown according to the law $N(\hat{\theta}_t^\lambda, \sigma^2(B_t^\lambda)^{-1})$ and to choose the action A_t that verifies : $a_t = argmax_{a \in A_t} a^T \tilde{\theta}_t^\lambda$.

The posterior at time t is a normal law $N(\hat{\theta}_t^\lambda, \sigma^2(B_t^\lambda)^{-1})$.

```
def get_action(self, arms):
    theta = np.random.multivariate_normal(self.mu, self.cov)
    theta /= np.linalg.norm(theta)
    return arms[np.argmax(np.dot(arms, theta))]
```

2.3 Model Comparison

To compare the three algorithms implemented, we created a Linear Bandit environment with $K = 10$ arms in dimension $d = 5$, and with Gaussian noise of variance 1. We plot the cumulative regret for the three methods with different parameters (ϵ for LinEpsGreedy and σ the parameter given for the calculation of $\beta(t, \delta)$). Figure 3 shows two experiments for $T = 100, N_{mc} = 10$ and $T = 1000, N_{mc} = 50$.



(a) $T = 100, N_{mc} = 10$ **(b)** $T = 1000, N_{mc} = 50$

Figure 3: Cumulative Regret for LinGreedy ($\epsilon = 0$), LinEpsGreedy ($\epsilon = 0.1$), LinUCB ($\lambda = 1, \delta = 0.05, \sigma = (1, 0.1, 4)$) and LinTS ($\lambda = 1, \sigma = 1$)

Although LinEspGreedy ($\epsilon = 0.1$) has the same cumulative regret curve at the beginning of the experiment (low T), the regret becomes far worse than all other algorithms as T grows. LinUCB seems to be a better algorithm, working well with $\sigma = 1$, which is the same value as the Gaussian noise. LinUCB with lower $\sigma = 0.1$ has the same performance. Hence the choice of σ does not seem to affect the result, except when it is too large. Indeed, LinUCB with $\sigma = 4$ leads to worse regret (curve close to LinEpsGreedy($\epsilon = 0.1$) in 3a). However, the algorithm that has the best performance in this experiment is LinTS, which regret is clearly lower in both 3a and 3b.

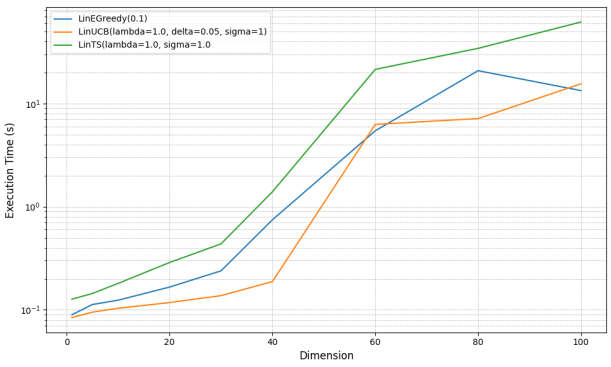


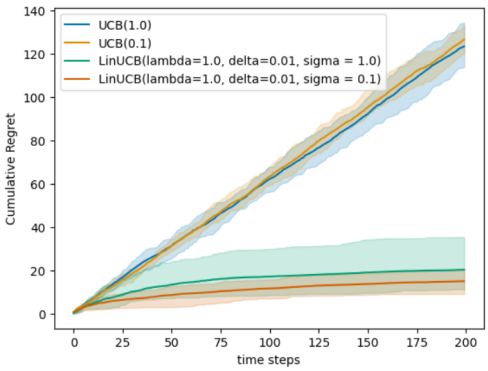
Figure 4: Mean Execution time as a function of d

We ran tests with different values of K and d which leads to the same conclusion that LinTS outperforms the other methods when looking at the cumulative regret. The adaptive posterior sampling is well effective in this problem where exploration is needed (as we said, low dimension and random action set). Our final experiment was to compare the mean execution times of LinEpsGreedy, LinUCB and LinTS in a simple problem ($K = 10 arms$), but with varying dimension d. In Figure 4, we can observe that although LinTS has the best cumulative regret, it is also the algorithm that converges the slower.

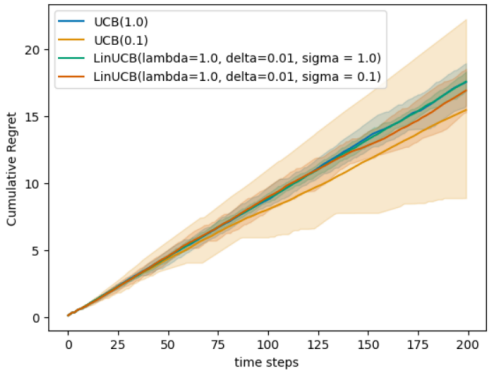
3 Bonus section: The role of the action set

The action sets can be 'arbitrary'. This means that, in principle, they do not have to follow a distribution, and they do not have to be random either. The action set can even be fixed. In order to test how certain sets of actions can be hard for LinUCB, we compute the simple UCB algorithm and compare the performances of LinUCB and UCB on various fixed sets. If we choose the number of arms K, their dimension d and the number of actions at random, most of the time we observe a better performance from LinUCB, as shown in 5a.

However, as presented in the article *The End of Optimism* (2017, Lattimore et al.), certain sets of actions can be hard. In particular, if $K = d$ and the set of actions is a base of R^d, LinUCB has a linear cumulative regret and UCB performs better. In 5b, we take the set of actions to be the columns of Id_d.



(a) Cumulative Regret of UCB and LinUCB for a random set of actions



(b) Cumulative Regret of UCB and LinUCB for a bad set of actions (Id_d)

Figure 5

We observe that LinUCB has indeed a linear cumulative regret on this example, proving that it performs badly on certain simple sets of action. UCB performs as good as LinUCB here.