Stochastic Linear Bandits - An Empirical Study

Darius Dabert darius.dabert@polytechnique.edu
Elia El Khoury eliaelkhoury2@gmail.com
Lecturer: Claire Vernade - website

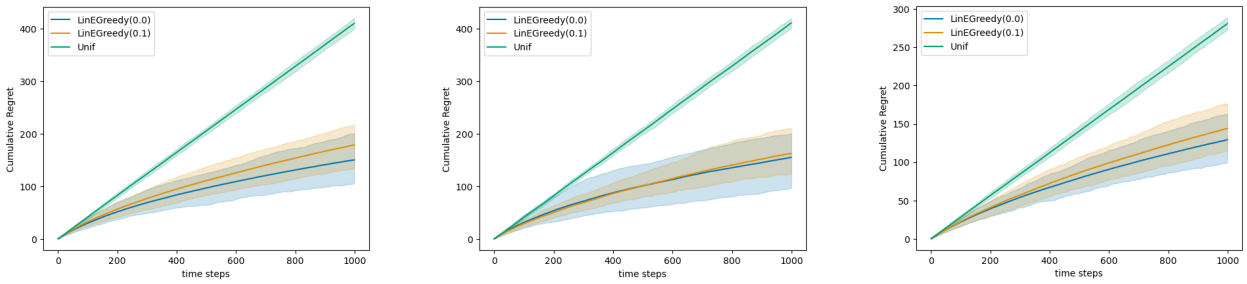
Problem 1: Linear Epsilon Greedy

Question 1

We start by completing the implementation of the Linear Bandit Environment and the Action Generation function. This function returns an array of K vectors uniformly sampled on the unit sphere in $\mathbb{R}^d$. We do so in order to satisfy the bounding hypothesis on the arm x. This is necessary to define the bound in UCB algorithms. After that, we implement the Linear Epsilon Greedy Algorithm and test it on a simple problem.

Question 2

To test our code, we use the following benchmark: it consists three problems with different ratio of $\frac{d}{K}$. For each, we have 30 (resp. 20, resp. 10) arms with a 10 (resp. 20, resp.10) dimensions vector θ each. These vectors are drawn randomly from a uniform distribution. We take a horizon of 1000 steps and run the algorithm 100 times. We use $\epsilon = 0$ and $\epsilon = 0.1$. The results of these tests are shown in figure 1a. First, we see that the Unif algorithm have a linear regret in average, which is expected since the distribution of arms is uniform. Then we see that the regret for LinEGreedy is a concave function of the time step, which means that we are learning the parameter theta and making better decision. We can see that if the ratio $\frac{d}{K}$ is different from 1, the smaller ϵ is, the faster the algorithm converges (meaning that the increasing of the regret is smaller). This baseline is a good baseline since the ϵ- greedy algorithm learns the reward parameter.



(a) $d = 10, K = 30$ **(b)** $d = 20, K = 20$ **(c)** $d = 30, K = 10$

Question 3

In the Linear ϵ-greedy algorithm, we have to compute a matrix inversion to determine the reward at each step. In the current version of the algorithm, we use the pinv function of the numpy library to do this matrix invesrion and it has a complexity of $O(d^3)$ where d is the dimension of each

arm. Considering the configuration of the problem, this complexity is not optimal and could be ameliorated. Indeed, we can use an incremental update method to inverse the matrix, like the Shermann-Morrison method which has a complexity of $O(d^2)$ (it only consist of matrices product from already computed quantities). This method suggests to write the update inverse matrix as

$$(B_{t+1}^{\lambda})^{-1} = (B_t^{\lambda} + x_{t+1}x_{t+1}^T)^{-1} = (B_t^{\lambda})^{-1} - \frac{(B_t^{\lambda})^{-1}x_{t+1}x_{t+1}^T(B_t^{\lambda})^{-1}}{1 + x_{t+1}^T(B_t^{\lambda})^{-1}x_{t+1}}.$$

The figure 2 shows the gain in runtime that we observe when using this better performing method.

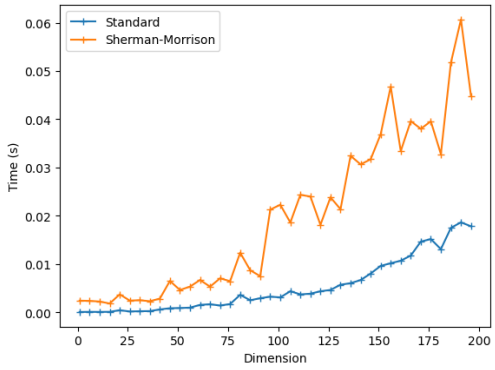


Figure 2: Comparison of Runtime for different matrix Invesrion methods. The test configuration was K = 200 arms, dimension varying from 5 to 200 and we averaged the runtime over a 100 inversions.

Problem 2: LinUCB and LinTS

Question 1

We start by implementing LinUCB as seen in class with its upper bound computed using the β-function presented in the lectures. We then implement LinTS which is based on a Bayesian approach and is characterized by its posterior for each time step t. In particular, we give the code for these two methods in the figures 3 and 4.

Question 2

Given a Gaussian prior on θ with mean $\mathbf{0}$ and covariance $\kappa^2 I$, and observed actions $x_1, \ldots, x_t$ and rewards $r_1, \ldots, r_t$, the posterior distribution of θ at the beginning of round $t+1$ is:

$$\theta \mid x_1, \ldots, x_t, r_1, \ldots, r_t \sim \mathcal{N}(\mu_t, \Sigma_t),$$

where:

$$\Sigma_t = \sigma^2 \left(\lambda I + \sum_{i=1}^{t} x_i x_i^{\top} \right)^{-1}, \quad \mu_t = \Sigma_t \sum_{i=1}^{t} x_i r_i \quad \text{and} \quad \lambda = \frac{\sigma^2}{\kappa^2}$$

```python
def _compute_ucbs(self, K, arms):
    ucbs = np.zeros(K)
    beta = (
        self.sigma * np.sqrt(2 * np.log(1 / self.delta) + self.d * np.log(1 + self
            .t / (self.d * self.lambda_reg)))
        + np.sqrt(self.lambda_reg) * np.linalg.norm(self.theta_hat)
    )

    for k in range(K):
        action = arms[k]
        norm_action = np.sqrt(action.T @ self.invcov @ action)
        ucbs[k] = (action.T @ self.theta_hat).item() + norm_action * beta

    chosen_arm_id = np.argmax(ucbs)
    chosen_action = arms[chosen_arm_id]

    return chosen_action
```

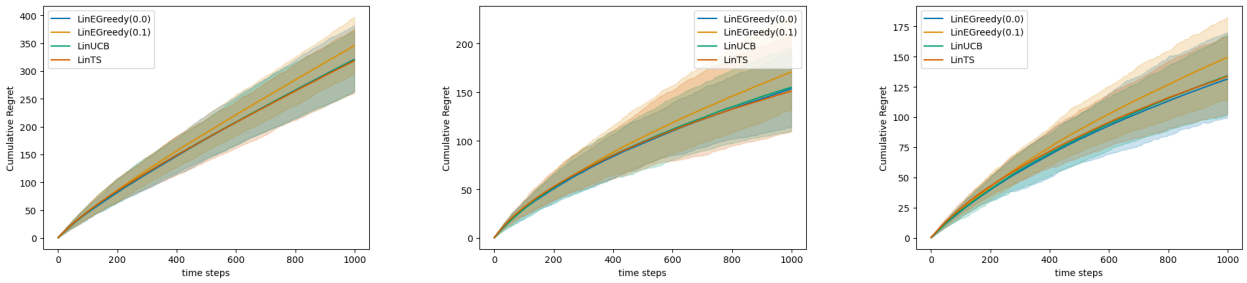
Figure 3: Code for UCB Computation

```python
def _compute_posterior(self, K, arms):
    self.theta_tilde = np.random.multivariate_normal(
        mean=self.theta_hat.flatten(),  # Posterior mean
        cov=(self.sigma ** 2) * self.invcov  # Posterior covariance
    ).reshape(-1, 1)

    # Calculate rewards
    rewards = arms @ self.theta_tilde

    chosen_arm_id = np.argmax(rewards)
    chosen_action = arms[chosen_arm_id]

    return chosen_action
```

Figure 4: Code for Posterior Sampling

Question 3

We tested the above algorithms on a series of values of d and K and there was no "clear winner" between them. In fact, the Linear ϵ-greedy algorithm with $\epsilon = 0.1$ tends to converge a little slower than the other three algorithms. On the other hand, LinUCB, LinTS and Lin ϵ-greedy with $\epsilon = 0$ have comparable performances. The figure 5 shows the performance of all the algorithms on the baseline used in Problem 1. As we take deviation into account, it seems there is no clear winner between the three algorithms.

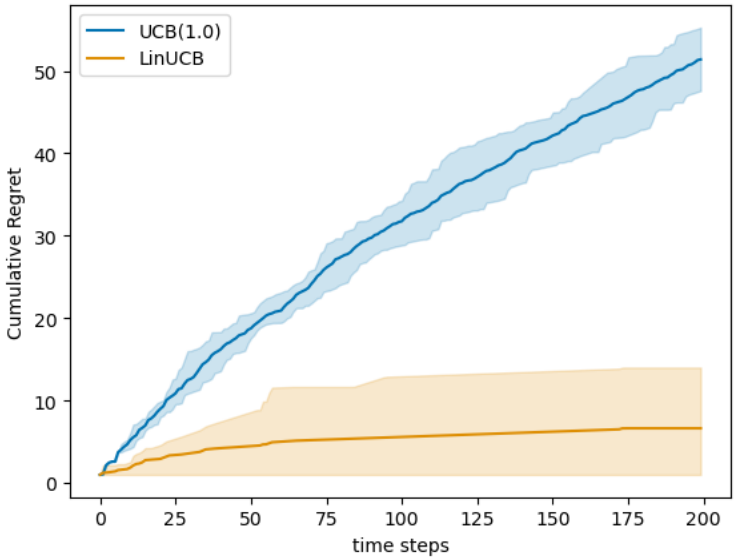


(a) $d = 10, K = 30$ **(b)** $d = 20, K = 20$ **(c)** $d = 30, K = 10$

Figure 5: Comparison of regret for various values of d and K.

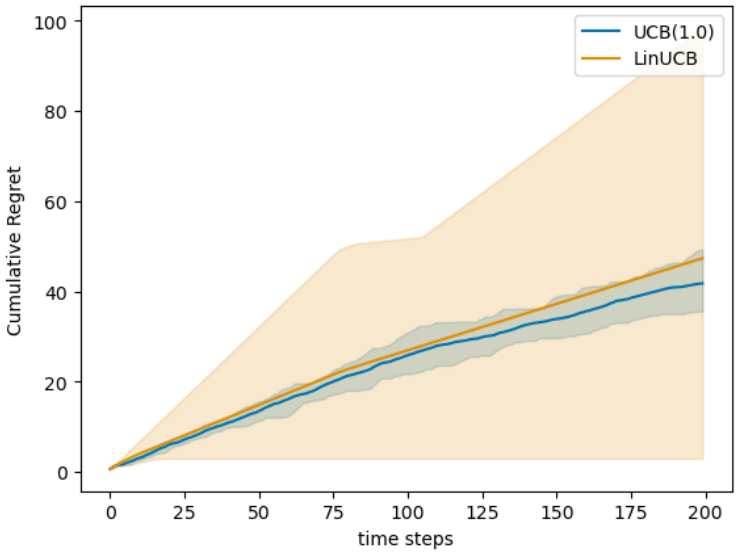
3. Bonus Part

LinUCB vs UCB Performance



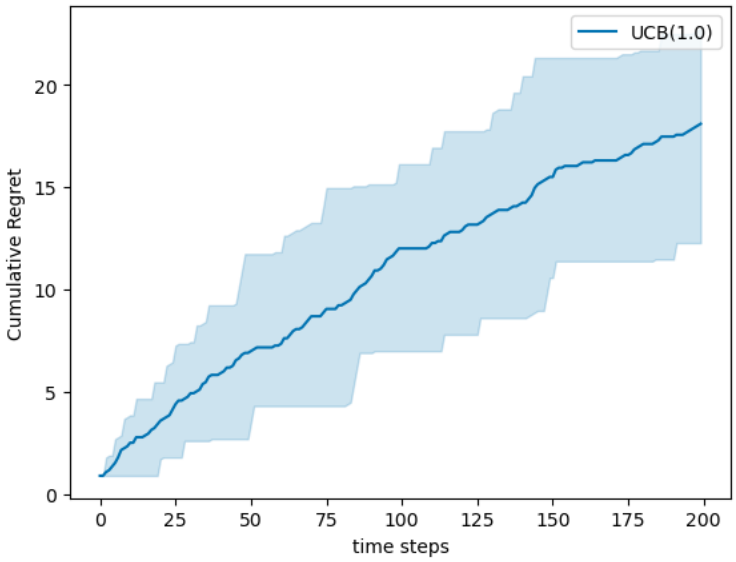
In this configuration (7 arms in dimension 3 with randomly generated but fixed arms), LinUCB significantly outperforms UCB. This difference arises due to the spread of information in LinUCB, which leverages contextual data to update estimates more effectively. This result highlights the advantage of contextual bandits when rich feature information is available.

Performance with Equal Arms and Dimension



This graph demonstrates that when the number of arms equals the dimension of the context space, the performances of LinUCB and UCB are similar. This suggests that under such configurations, the benefit of context in LinUCB does not provide a significant advantage over UCB.

Cumulative Regret with Specific Action Configuration



This configuration (describe below) illustrates the cumulative regret of UCB in a specific action configuration defined as:

$$\text{fixed\_actions} = \{[1, 0], [0, 1], [1 - \epsilon, \alpha \cdot \epsilon]\}.$$

As shown, the regret is linear, corroborating findings in *The End of Optimism* by Lattimore et al. This behavior underscores the limitations of UCB in certain non-standard action settings where linear regret emerges due to suboptimal exploration.

These graphs collectively highlight how algorithm performance can vary significantly with problem configuration. The insights emphasize the importance of adapting algorithms to the structure of the problem.

References

- Lattimore, T., Szepesvári, C. (2016). *The End of Optimism? An Asymptotic Analysis of Finite-Armed Linear Bandits*. Retrieved from https://arxiv.org/pdf/1610.04491