Stochastic Linear Bandits An Empirical Study

Students: Oliver JACK, Paulo SILVA
Lecturer: Claire Vernade

1 Problem 1

In a first step, we implemented the contextual Linear Bandit environment, which is entirely defined by an unknown regression vector $\theta_* \in \mathbb{R}^d$, a set of K arms (actions) $\mathcal{X}_t \subseteq \mathbb{R}^d$, and the variance σ^2 of a centered noise η_t. In case the K actions are not fixed by the user, the method `get_action_set` calls the function `ActionsGenerator` to generate K d-dimensional random vectors $x_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_d)$, which are subsequently normalized to unit length. Finally, the `get_reward` method computes the resulting reward after having chosen the arm $x_t \in \mathcal{X}_t$, given by the formula

$$r_t = \langle \theta_*, x_t \rangle + \eta_t. \tag{1}$$

For our computational experiments, we decided to choose (unless specified differently) the values $d = 20$, $K = 50$, $N = 100$ (number of Monte Carlo simulations) and $T = 1000$ (finite horizon), while the random seed is set to 42 for reproducibility purposes.

Next, we implemented the Linear ϵ-Greedy class, a policy which selects a random action vector from the action set with a probability ϵ, otherwise it chooses the arm x_t which maximizes the estimated reward $\hat{r}_t = \langle \hat{\theta}_t, x_t \rangle$, where $\hat{\theta}_t$ is the regularized least-squares method of the unknown regression vector θ_*. At each timestep $t \leq T$, the estimate $\hat{\theta}_t$ is updated according to the rule

$$\hat{\theta}_t = (A_t^\lambda)^{-1} b_t \tag{2}$$

where $A_t^\lambda = \lambda \mathbf{I}_d + \sum_{s=1}^t x_s x_s^T$ and $b_t = \sum_{s=1}^t r_s x_s$. Here, A_t^λ is the regularized design matrix (initialized to $\lambda \mathbf{I}_d$ with regularization parameter $\lambda = 1$), while b_t (initialized to $\mathbf{0}_d$) accumulates the weighted rewards.

As one can imagine, this policy largely depends on the choice of the hyperparameter ϵ. To evaluate the performance of our policy for different values of ϵ, we can use the cumulative regret as comparison metric:

$$R(T) = \sum_{t=1}^{T} (r_t^* - r_t), \tag{3}$$

where r_t^* is the reward of the optimal action at time t and r_t is the reward of the action chosen by the algorithm.

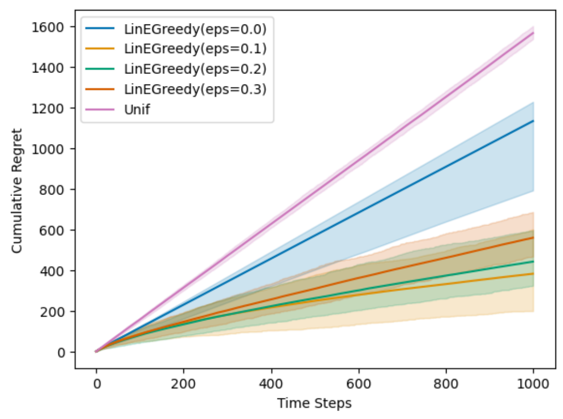
With the help of this chosen benchmark, we compared the Linear ϵ-Greedy policy for the values $\epsilon = 0, 0.1, 0.2, 0.3$ against a uniform random policy, which uniformly selects a random arm from the action space. Based on the plot (1a), one can observe that the regret grows relatively steeply for $\epsilon = 0$. This can be explained by the fact that the algorithm always exploits, which causes it to prematurely focus on suboptimal arms due to insufficient exploration. Meanwhile, for the values $\epsilon = 0.1, 0.2, 0.3$, the trade-off between exploration and exploitation seems to improve the performance. Furthermore, the plot shows that finding the right balance between both is key, as in this particular case excessive exploration leads to suboptimal performance ($\epsilon = 0.1$ achieves the lowest regret). Finally, the cumulative regret grows the fastest for the uniform policy, since it is entirely exploratory and doesn't exploit the best observed actions.

Linear ϵ-Greedy provides a strong baseline for contextual bandit problems but is not necessarily the best-performing algorithm (see problem 2 for LinUCB and LinTS), especially in dynamic or high-dimensional environments. The results demonstrate the importance of balancing exploration and exploitation, with ϵ being a good choice for the selected benchmark.

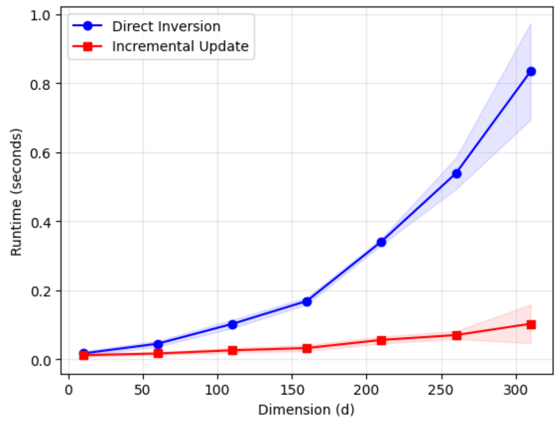
The computational bottleneck of the Linear ϵ-Greedy update rule comes down to inverting the design matrix A_t^λ, which has a time complexity of $\mathcal{O}(d^3)$. To reduce this cost, the matrix $(A_t^\lambda)^{-1}$ can be approximated efficiently by adopting the Sherman-Morrison formula Sherman and Morrison (1950), which incrementally updates the matrix $(A_t^\lambda)^{-1}$:

$$(A_{t+1}^\lambda)^{-1} = (A_t^\lambda)^{-1} - \frac{(A_t^\lambda)^{-1} x_t x_t^T (A_t^\lambda)^{-1}}{1 + x_t^T (A_t^\lambda)^{-1} x_t} \tag{4}$$

This allows us to reduce the computational time complexity of the update step to $\mathcal{O}(d^2)$. This increasing difference in runtime can be observed in plot (1b). Hence, for a small precision price, the runtime can be improved significantly as the dimension d grows.



(a) Cumulative regret for LinEpsGreedy($\epsilon = 0, 0.1, 0.2, 0.3$) & Uniform



(b) Direct Inversion vs. Incremental Update: Average runtime wrt. dimension d

Figure 1: Figures related to problem 1

2 Problem 2

While the Linear ϵ-Greedy policy is a good first approach to tackle contextual linear bandits, there are other policies that explicitly balance exploration and exploitation without relying on random exploration. In this section, we analyze two such policies: LinUCB and LinTS.

LinUCB uses an optimistic approach to select the next arm by choosing the action vector x_t^a which maximizes the upper confidence bound on the estimated reward:

$$\text{UCB}_t^a = \langle \hat{\theta}_t, x_t^a \rangle + \delta\sigma\sqrt{(x_t^a)^T (A_t^\lambda)^{-1} x_t^a} \tag{5}$$

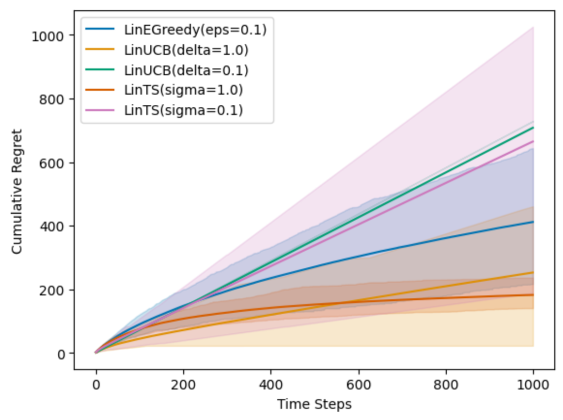
where $\hat{\theta}_t$, A_t^λ and b_t are all updated in the same way as in the ϵ-Greedy case. δ is a hyperparameter controlling the width of the confidence interval, while σ^2 represents the variance of the noise. By adding this uncertainty term to the estimated reward, a certain level of exploration is guaranteed, while still exploiting the best-known actions.

LinTS adopts a Bayesian approach by sampling in each round t a regression vector $\tilde{\theta}_t \sim \mathcal{N}(\hat{\theta}_t, \sigma^2(A_t^\lambda)^{-1})$, which is the posterior distribution at time t. Here, $\hat{\theta}_t$ and A_t^λ are as defined previously. The algorithm then selects the action vector x_t^a maximizing the sampled reward $\langle \tilde{\theta}_t, x_t^a \rangle$. Generally speaking, LinTS aims to balance exploration and exploitation by sampling plausible parameter estimates from the posterior distribution, allowing for exploration without requiring an explicit confidence parameter like δ for LinUCB.

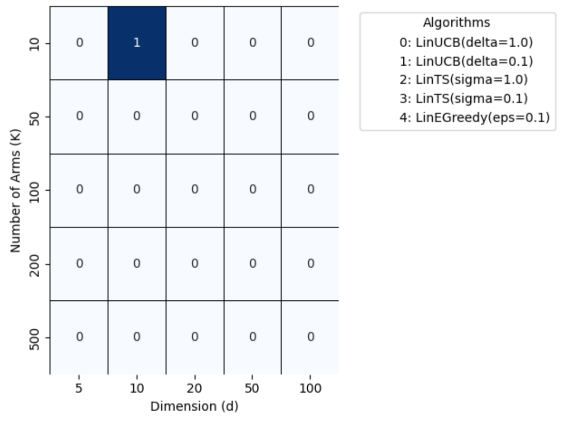
When plotting the evolution of the cumulative regret of the different algorithms for a specific initial configuration (see plot (2a)), one can see that both LinUCB and LinTS perform relatively well for some δ and σ, while less so for others. This sensitivity shows that the interpretation is heavily dependent on the considered hyperparameters δ and σ for LinUCB and LinTS, as was also the case for LinEpsilonGreedy. Furthermore, the plot clearly highlights that the chosen finite horizon plays an important role, as in this particular case LinUCB($\delta = 1$) tends to outperform LinTS($\sigma = 1$) up until timestep ≈ 600, before LinTS overtakes LinUCB and maintains a lower cumulative regret for the remainder of the horizon.

Finally, to evaluate which algorithm performs best based on the initial configuration, we designed an experiment comparing `LinEpsilonGreedy(`$\epsilon = 0$`)`, `LinEpsilonGreedy(`$\epsilon = 0.1$`)`, `LinUCB(`$\delta = 1$`)` and `LinTS(`$\sigma = 1$`)`, for different pairs of (K, d) ($K = 10, 50, 100, 200, 500; d = 5, 10, 20, 50, 100$). For each (K, d) pair, we decided to run 100 random simulations with different seeds, computing the cumulative regret for each algorithm after a finite horizon of $T = 1000$, deeming the one with the lowest average regret as the "optimal" policy for that configuration. To visualize the results, we created a heatmap where each cell represents a (K, d) pair and the color indicates the algorithm that achieved lowest average cumulative regret after T steps.

Based on the obtained heatmap (2b), LinUCB (here especially with $\delta = 1$) tends to consistently outperform the other algorithms when it comes to minimizing the cumulative regret for various initial configurations, showing its robustness across varying conditions. However, it is noteworthy that this experiment largely depends on the chosen hyperparameters for the policies, as well as the finite horizon T and the number of random seeds N. Due to the high number of parameters that need to be fine-tuned, determining the "best" algorithm is a highly difficult or even impossible context-dependent task.



(a) Cumulative regret for LinUCB($\delta = 1$) & LinTS($\sigma = 1$)



K \ d	5	10	20	50	100
10	0	1	0	0	0
50	0	0	0	0	0
100	0	0	0	0	0
200	0	0	0	0	0
500	0	0	0	0	0

(b) "Optimal" algorithm for each (K, d) pair based on cumulative regret

Figure 2: Figures related to problem 2

References

Lattimore, T. and Szepesvari, C. (2017). The end of optimism? an asymptotic analysis of finite-armed linear bandits. In *Artificial Intelligence and Statistics*, pages 728–737. PMLR.

Sherman, J. and Morrison, W. J. (1950). Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Ann. Math. Statist.*, 21(4):124–127.

A Bonus Section

In this bonus section, we investigate how the structure of the action set has an impact on the performance of LinUCB and standard UCB algorithms. Our experiment builds on the ideas from Lattimore and Szepesvari (2017), who were able to show that certain action sets can pose problems for LinUCB, even leading to linear regret in specific cases.

Let us start off by considering the case when the action set is randomly generated. As can be seen in figure (3), LinUCB typically outperforms standard UCB. This can be explained by the fact that LinUCB tends to learn the linear relationships between actions and the unknown parameter θ to find a good balance between exploration and exploitation.

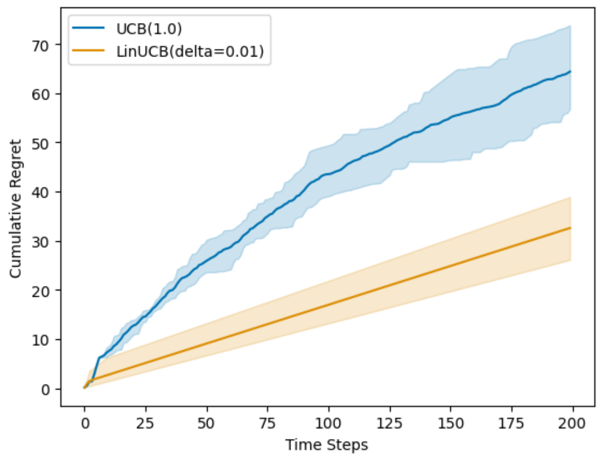


Figure 3: LinUCB vs. UCB for arbitrary action sets

On the other hand, let us introduce a counterexample of a fixed action set, inspired by Lattimore and Szepesvari (2017), for which LinUCB struggles in terms of cumulative regret. Set $d = 2$ and consider 3 well-defined actions $e_1 = (1, 0)$, $e_2 = (0, 1)$ and $x = (1 - \epsilon, 8\alpha\epsilon)$, where ϵ (set to 0.1 here) takes small values and α (set to 1 here) is a parameter linked to the algorithm's confidence bound. The true (in practice unknown) θ_* is set to e_1. The figure (4) shows us that LinUCB performs poorly with respect to UCB in this setting, with the cumulative regret growing linearly. The intuition behind this is that LinUCB prematurely focuses on e_1 and x, ignoring e_2 because it appears to be less relevant with respect to θ. However, this oversight leads to LinUCB missing out on valuable information provided by e_2, negatively impacting its overall performance.

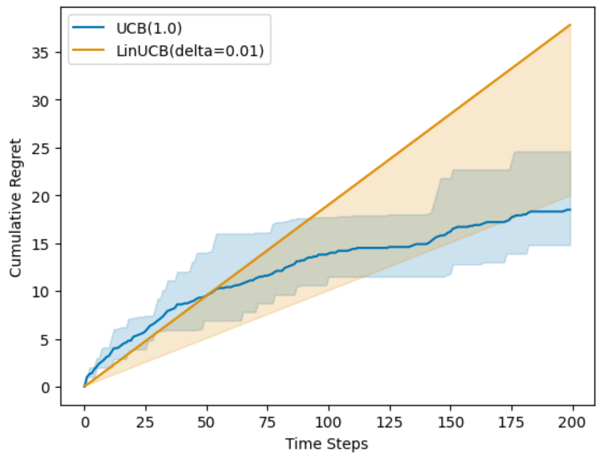


Figure 4: LinUCB vs. UCB for counterexample action set

Based on our experiments, we can conclude that LinUCB is highly effective when the action set is well-aligned with θ. However, the algorithm's performance deteriorates in scenarios where the action set is sparse or poorly-aligned with θ, as in the case of our counterexample. In contrast, standard UCB is less sensitive to the structure of the action set but often performs worse in contextual linear bandit environments.

B Code

The full version of our code, including all generated plots, is available in our GitHub repository linked down below:

https://github.com/olijacklu/MVA/blob/main/Reinforcement%20Learning/Projects/Linear_Bandit.ipynb