Stochastic Linear Bandits An Empirical Study

Students: Marion Chabrol and Suzie Grondin
Lecturer: Claire Vernade

Problem 1 : Linear Epsilon Greedy

We tested the problem using an action space of size $K = 7$ on $d = 3$ dimensions, over 10 Monte Carlo simulations, each spanning 40 steps. The tests were conducted in two environments: an i.i.d. environment and a fixed environment with $\sigma = 1$. Below is the code for the action selection in LinEpsilonGreedy:

```python
def get_action(self, arms):
    if self.epsilon_decay is not None:
        self.eps = self.epsilon_decay(self.t, self.eps)
    K, _ = arms.shape
    u = np.random.rand()
    if u < self.eps: # With proba epsilon
        return arms[np.random.choice(K)] # Exploration: choose a random action
    else:
        # Exploitation: choose the action with maximum estimated reward
        estimated_rewards = arms @ self.hat_theta # empirical mean
        return arms[np.argmax(estimated_rewards)]
```

The results in Fig. 1a show that, in both i.i.d. and fixed environments, the cumulative regret of LinEpsGreedy(0.0) and LinEpsGreedy(0.1) grows much slower (logarithmically) compared to LinUnif, whose regret grows linearly. This is because LinUnif selects actions randomly, ignoring past results, while LinEpsGreedy balances exploration and exploitation: LinEpsGreedy(0.0) fully exploits past data, and LinEpsGreedy(0.1) introduces limited exploration, enabling better adaptation to the environment.

Notably, in i.i.d. environments, the inherent randomness ensures implicit exploration, which is why even greedy approaches like LinEpsGreedy(0.0) perform well. Initially, LinEpsGreedy(0.1) outperforms LinEpsGreedy(0.0), but after 20–30 steps, the trend reverses as pure exploitation becomes more effective once the environment is well-understood. Note that the environment here is relatively small, which allows greedy to outperform, but with a larger K or larger σ^2, this would not necessarily be the case. We'll come back to this in more detail in Problem 2. We also conducted experiments with decreasing exploration (ϵ) using different schedules (linear, exponential and customed decay), which further improved the cumulative regret, as shown in Fig. 1b 1b. For reasons of robustness and simplicity, we use LinEpsGreedy(0.1) as our baseline.

The direct inversion of a covariance matrix $\mathbf{B}_t$ of size $d \times d$ involves solving linear equations or performing operations like Gaussian elimination, resulting in a complexity of $\mathcal{O}(d^3)$. This high complexity arises because matrix inversion requires processing all elements multiple times through elimination and substitution.

To address this, we use the Sherman-Morrison formula:

$$\mathbf{B}_t^{-1} = \mathbf{B}_{t-1}^{-1} - \frac{\mathbf{B}_{t-1}^{-1}\mathbf{x}\mathbf{x}^\top\mathbf{B}_{t-1}^{-1}}{1 + \mathbf{x}^\top\mathbf{B}_{t-1}^{-1}\mathbf{x}}.$$

This approach enables incremental updates of $\mathbf{B}_t^{-1}$, avoiding the need to recompute the inverse from scratch. The operations primarily involve vector-matrix multiplications.

As a result, the complexity of each incremental update is reduced to $\mathcal{O}(d^2)$, which is significantly faster than the naive $\mathcal{O}(d^3)$ approach. The runtime improvements as a function of d are evident in Fig. 1c, where the incremental update consistently outperforms direct inversion, particularly for larger values of d.

Problem 2: LinUCB and LinTS

Below is the code for the action selection in LinUCB:

```python
def get_action(self, arms):
    K, _ = arms.shape
    beta_t = self._beta_t()
    ucb_values = []
    for arm in arms:
        # expected reward (exploitation) +  confidence interval (exploration)
        ucb = np.dot(arm, self.hat_theta) + beta_t * np.sqrt(np.dot(arm, np.dot(self.invcov, arm)))
        ucb_values.append(ucb)
    return arms[np.argmax(ucb_values)] # Choose the action with the highest UCB
```

Below is the code for the action selection in LinTS:

```python
def get_action(self, arms):
        # Sample a parameter vector from the posterior distribution
        sampled_theta = np.random.multivariate_normal(self.hat_theta, self.invcov)
        estimated_rewards = np.dot(arms, sampled_theta)
        # Choose the action with the highest estimated reward
        return arms[np.argmax(estimated_rewards)]
```

The prior on θ is Gaussian: $\theta \sim \mathcal{N}(\mathbf{0}, \lambda I)$. The prior density is: $P(\theta) \propto \exp\left(-\frac{1}{2}\theta^\top \left(\frac{1}{\lambda} I_d\right) \theta\right)$.

$$
\begin{aligned}
P(Y_1, \ldots, Y_t \mid \theta) &\propto \exp\left(-\frac{1}{2\sigma^2} \sum_{s=1}^{t} (Y_s - A_s^\top \theta)^2\right). \\
&\propto \exp\left(-\frac{1}{2\sigma^2} \left[\sum_{s=1}^{t} Y_s^2 - 2\sum_{s=1}^{t} Y_s A_s^\top \theta + \theta^\top \left(\sum_{s=1}^{t} A_s A_s^\top\right) \theta\right]\right) \\
&\propto \exp\left(-\frac{1}{2\sigma^2} \left[\theta^\top \mathbf{X}_t \theta - 2\mathbf{y}_t^\top \theta\right]\right) \text{ with } \mathbf{X}_t = \sum_{s=1}^{t} A_s A_s^\top, \quad \mathbf{y}_t = (\sum_{s=1}^{t} Y_s A_s^\top)^\top
\end{aligned}
$$

$$
\begin{aligned}
\text{Using Bayes' rule : } P(\theta \mid \{A_s, Y_s\}_{s=1}^t) &\propto P(Y_1, \ldots, Y_t \mid \theta) P(\theta) \\
&\propto \exp\left(-\frac{1}{2}\left[\theta^\top \left(\frac{1}{\sigma^2}\mathbf{X}_t + \frac{1}{\lambda} I_d\right)\theta - 2\frac{1}{\sigma^2}\mathbf{y}_t^\top \theta\right]\right) \\
&\propto \exp\left(-\frac{1}{2}\left[\theta^\top B_t \theta - 2 b_t^\top \theta\right]\right) \text{ with } B_t = \frac{1}{\sigma^2}\mathbf{X}_t + \frac{1}{\lambda} I_d, \quad b_t = \frac{1}{\sigma^2}\mathbf{y}_t \\
&\propto \exp\left(-\frac{1}{2}\left[(\theta - \hat{\theta}_t)^\top B_t (\theta - \hat{\theta}_t) - \hat{\theta}_t^\top B_t \hat{\theta}_t\right]\right) \text{ where } \hat{\theta}_t = B_t^{-1} b_t
\end{aligned}
$$

Hence, for Thomson sampling, the posterior distribution at time t is :

$$\boxed{\theta \mid \{A_s, Y_s\}_{s=1}^t \sim \mathcal{N}(\hat{\theta}_t, B_t^{-1})}$$

where $\hat{\theta}_t = B_t^{-1} b_t$, $\quad b_t = \frac{1}{\sigma^2}\sum_{s=1}^t Y_s A_s$, $B_t = \frac{1}{\sigma^2}\sum_{s=1}^t A_s A_s^\top + \frac{1}{\lambda} I_d$

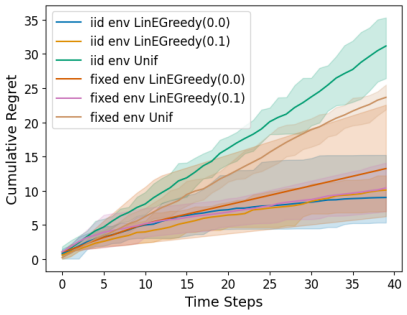
We conducted a first experiment to evaluate performance in a large action space ($K = 1000$ arms) with $d = 3$, over 10 Monte Carlo simulations of 10,000 steps in an i.i.d. environment ($\sigma = 1$), as shown in Fig. 2a. LinEGreedy performs the worst due to inefficient random exploration, leading to high cumulative regret. **LinTS performs the best in large action space**, balancing exploration and exploitation effectively with adaptive posterior sampling while remaining computationally scalable. LinUCB, while better than LinEGreedy, is less efficient than LinTS, as computing confidence bounds for all actions is costly. Fig. 2b shows LinUCB's computation time increases much faster than LinTS and LinEGreedy as K varies between 5 and 5000. However, in very high-dimensional spaces, LinTS's posterior updates become expensive, so

LinUCB is expected to perform better in very high-dimensional spaces, as its confidence bounds guide exploration more efficiently.

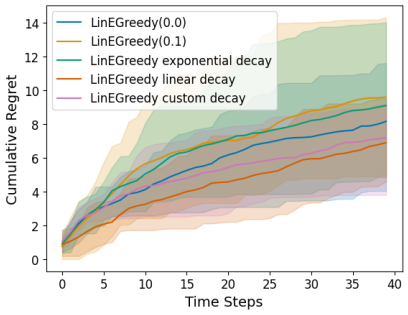
We also compared performance in high-noise ($\sigma = 4$) and low-noise ($\sigma = 0.25$) environments, using $K = 7$ arms and $d = 3$ over 10 Monte Carlo simulations of 10,000 steps (Fig. 2c). **LinTS excels in noisy settings**, leveraging posterior uncertainty for effective exploration-exploitation without explicit confidence bounds. **LinUCB performs best in low-noise environments**, where tight bounds focus on exploitation, but struggles in high noise due to overly conservative bounds, leading to excessive exploration and slow convergence. LinEGreedy consistently underperforms, as it ignores reward structure and uncertainty.

In stationary settings, we expect LinTS to perform best, as it naturally balances exploration and exploitation without tuning. LinUCB should also work well but may require careful tuning of β_t. LinEGreedy is likely inefficient, wasting rounds on random exploration instead of exploiting the stationary structure. Conversely, **in adversarial or non-stationary settings, LinUCB is expected to perform well**, targeting uncertain regions effectively with its optimism-based exploration. LinTS may struggle, as its posterior adapts slowly to changes, and LinEGreedy remains ineffective, lacking focus and robustness in dynamic environments.

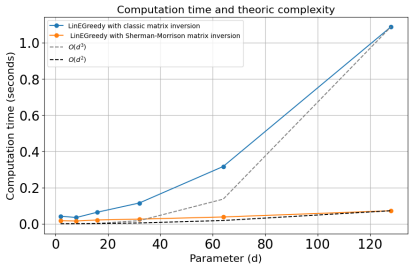
In summary, **each algorithm is suited to specific environments**. While there is no universal winner, understanding the environment is crucial to selecting the best approach.



(a) Comparison of cumulative regret for LinEpsGreedy(0.0), LinEpsGreedy(0.1), and LinUnif in both i.i.d. and fixed environments.

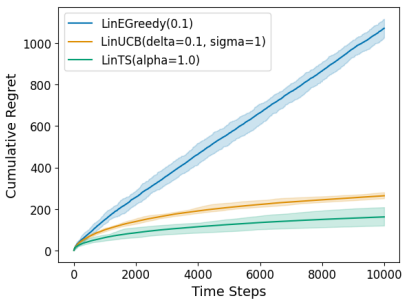


(b) Regret comparison between LinEpsGreedy(0.0), LinEpsGreedy(0.1) and decaying ϵ schedules.

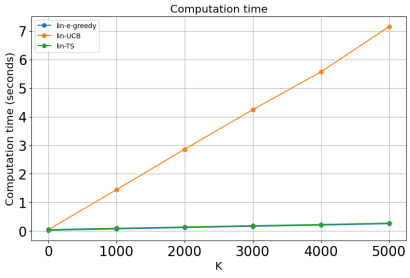


(c) Runtime comparison of direct matrix inversion and the incremental update using the Sherman-Morrison formula.

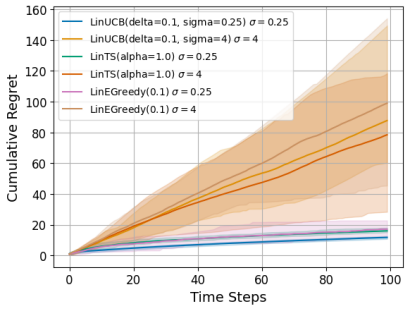
Figure 1: Study of the LinEGreedy algorithm caracteristics



(a) Regret comparison between LinEpsGreedy(0.1), LinUCB and LintTS for $K = 100$.



(b) Runtime comparison between LinEpsGreedy(0.1), LinUCB and LintTS.



(c) Regret comparison between LinEpsGreedy(0.1), LinUCB and LintTS for different σ.

Figure 2: Comparisons between LinEGreedy, LinUCB and LinTS