CSC 665 Spring 2020 HW2*

Problem 1. This problem is about anytime the Hoeffding's inequality.

(a) As a warmup, derive both versions of the inequalities. That is, let $\hat{\mu}_t = \frac{1}{t}\sum_{s=1}^t X_s$ where X_s is an i.i.d. ($\sigma^2 = 1$)-sub-Gaussian. Prove the following using first principles (i.e., do not use Hoeffding's bound + variable change):

$$\text{(fixed time)} \qquad \text{Fix } t \ge 1. \;\; \mathbb{P}\left(\hat{\mu}_t \ge \sqrt{\frac{2}{t}\ln(1/\delta)}\right) \le \delta$$

$$\text{(anytime)} \qquad \mathbb{P}\left(\forall t \ge 1, \hat{\mu}_t \ge \sqrt{\frac{2}{t}\ln(4t^2/\delta)}\right) \le \delta$$

(b) Your classmate claims that in fact the fixed time deviation bound above actually works for all time t throughout, with probability at least $1 - \delta$. Let us empirically show that she is wrong! Let $N = 10,000$ and $\delta = 0.1$. Draw N Gaussian random variables from mean 0 and variance 1; call them $X_1, \ldots, X_N$. With those, compute $\{\hat{\mu}_t\}_{t=1}^N$. Record whether there exists $t \in [1, N]$ such that $\hat{\mu}_t$ that cross the deviation $\sqrt{(2/t)\ln(1/\delta)}$. Let $Y = 1$ if this was true and 0 otherwise. Now, repeat this 100 times with a fresh set of random samples. Denote by $Y_1, \ldots, Y_{100}$ those binary values. If your friend is correct, we must be seeing that the average of $\{Y_i\}_{i=1}^{100}$ is around δ or less.

- Use your favorite programming language to perform the simulation.
- Report the average of $\{Y_i\}$ for both the fixed time version and anytime version.
- Pick some of the random trial and plot $t \cdot \hat{\mu}_t$ and the both deviation bounds multiplied by t, all three of them in one plot with x-axis being t. (Multiplying t is merely to improve the visual).
- Submit your code, plot, and explanations.

*v1

Problem 2. This is about UCB and LinUCB. In the class, we learned a fixed budget version of UCB where we need to feed n, the time horizon, to the algorithm. In this homework, let us use the anytime version of UCB, which selects arms by

$$A_t = \arg\max_{i \in \{1,\ldots,k\}} \hat{\mu}_i(t-1) + \sqrt{\frac{2\log(t^{2.1})}{T_i(t-1)}}$$

where we enforce the objective function to be ∞ when the count $T_i(t-1)$ is zero.

Recall that we assume that the arms $a \in \mathcal{A}$ satisfies that $\|a\|_2 \le 1$ and $\|\theta^*\|_2 \le 1$. Let us now relax the latter assumption to $\|\theta^*\|_2 \le S$ where $S > 0$ is known to us. My apologies for not being exact with the definition of $\sqrt{\beta_t}$ in the class, which contained $\|\theta^*\|_2$. This information is not know to the learner. Instead, we need to use:

$$\sqrt{\beta_t} = \sqrt{\lambda} \cdot S + \sqrt{\log\left(\frac{|V_{t-1}|}{|V_0| \cdot \delta^2}\right)}$$

For other details, please look at the lecture whiteboard shared in piazza (dropbox link).

(a) Implement UCB.

(b) Before implementing LinUCB, derive sequential updates so we have the per-time-step time complexity of $O(d^2)$ w.r.t. the dimension d. Specifically,

- Derive an update equation for V_t^{-1} based on V_{t-1}^{-1} directly (rather than computing the inversion). Use Woodbury matrix identity (see Wikipedia) and the fact that $V_t = V_{t-1} + A_t A_t^\top$.
- Derive an update equation for $|V_t|$ from $|V_{t-1}|$, without directly computing the determinant. Hints can be found somewhere in the lecture whiteboard shared in piazza (dropbox link).

(c) Implement LinUCB. Again, ensure that the per-time-step time complexity must be $O(d^2)$ w.r.t. d. Otherwise, points will be deducted.

(d) Design simulation setups where there are k arms, each with known feature vectors $a \in \mathbb{R}^d$. Compare UCB and LinUCB w.r.t. the cumulative regret. Suggestions:

- Design at least two settings: one where LinUCB might perform better and one where UCB might perform better.
- Submit your code, plots, and explanations.

Don't worry if your designed setting does not work as you expected; this is an open question. Just be sure to provide your thoughts in the answer.

Tip: I emphasized this in the last homework too, but please do spend some time to develop test cases, visually inspect your code, printout values, use step-by-step debugger to check values, etc., to convince yourself that the algorithm is correct. Mathematical code is hard to debug, but the cost of bugs is tremendous.