CS 747: Programming Assignment 1

Total marks: 12

(Prepared by Mohith and Dibyangshu)

This assignment tests your understanding of the regret minimisation algorithms discussed in class. You will implement and compare different algorithms for sampling the arms of a stochastic multi-armed bandit. To begin, in Task 1, you will implement epsilon-greedy exploration, UCB, KL-UCB, and Thompson Sampling, more or less identical to the versions discussed in class. Task 2 requires you to optimise the scaling coefficient of the exploration bonus in UCB. In both tasks 1 and 2, your algorithms will be tested on bandit instances that yield 0-1 rewards (the Bernoulli case). Tasks 3 and 4 require you to improvise to tackle a scenario in which the rewards come from a finite set contained in [0, 1], which generalises the Bernoulli case. In Task 3, you are encouraged to develop as efficient a regret-minimisation algorithm as possible for this scenario; the choice of how to do so is left to you. In Task 4, your algorithm has a different objective from (the usual) regret minimisation. In particular, the aim is to maximise the expected number of times the reward obtained exceeds a given threshold.

All the code you write for this assignment must be in Python 3.8.10. The only libraries you may use are Numpy v1.21.0 (to work with vectors and matrices) and Matplotlib (for generating plots). All of these come installed with the docker image that has been shared for the course.

Data and Scripts

This directory has two folders: `instances` and `submissions`; as well as a file named `outputFormat.txt`.

`instances` contains a separate folder for each of the tasks, within which relevant bandit instances are provided.

`submission` is the folder in which you must place all your submission-related material. It contains a script `check.sh` for verifying the input-output behaviour of your program, and a script `verifyOutput.py` for verifying that your submission data is printed in correct format and reproducible.

The file `outputFormat.txt` is provided to illustrate the format of output your code is expected to generate.

The program you create will have to take in several command-line arguments. By running the checking script provided, you can make sure that your program consumes the input parameters correctly and produces the desired output.

You will have to perform multiple random executions of different algorithms on the given instances and place all your data in a text file. The example provided shows you the expected format.

All your submission materials should go into the `submission` folder, which you will compress and submit.

Input format

Tasks 1, 2

For tasks 1 and 2, your code will be run on Bernoulli bandit instances encoded as text files with the mean rewards of the arms provided one to a line. Hence the number of lines gives the number of arms. The instances provided have 2, 5, and 25 arms.

Task 3, 4

For tasks 3 and 4, the bandit instances are again encoded as text files. The first line specifies the "support", which is the list of possible values the reward can take. For example, it could be `0 0.2 0.4 0.6 0.8 1`. The arms all have this same support, which means, when an arm is sampled, the reward will be one of these 6 values. The support will always be contained in [0, 1]. Assume it is of size m. If the number of arms is n, then the input file will have a total of n + 1 lines, wherein lines 2 through n + 1 give the probability distributions corresponding to the arms. These lines will also have m entries, again in space-separated format like the first line. Their m entries specify a probability distribution over the reward support. Suppose the support set is `0 0.2 0.4 0.6 0.8 1`; now, if in line 5, the third number is 0.16, then it means arm 4 gives a a reward of 0.4 with probability 0.16.

Problem statements for tasks

Task 1

In this first task, you will implement the sampling algorithms: (1) epsilon-greedy, (2) UCB, (3) KL-UCB, and (4) Thompson Sampling. This task is straightforward based on the class lectures. The instructions below tell you about the code you are expected to prepare: much of it will generalise to the other tasks, as well.

You will have to prepare a source file called `bandit.py` that is used for Task 1 as well as all the other tasks. You can decide for yourself how to modularise the code and name the internal functions. What we shall insist upon is the input-output behaviour of `bandit.py`, which we describe below.

`bandit.py` must accept the following command line parameters.

> `--instance in`, where `in` is a path to the instance file.
> `--algorithm al`, where `al` is one of `epsilon-greedy-t1`, `ucb-t1`, `kl-ucb-t1`, `thompson-sampling-t1`, `ucb-t2`, `alg-t3`, `alg-t4`.
> `--randomSeed rs`, where `rs` is a non-negative integer.
> `--epsilon ep`, where `ep` is a number in [0, 1]. For everything except epsilon-greedy, pass 0.02.
> `--scale c`, where `c` is a positive real number. The parameter is only relevant for Task 2; for other tasks pass `--scale 2`.
> `--threshold th`, where `th` is a number in [0, 1]. The parameter is only relevant for Task 4; for other tasks pass `--threshold 0`.
> `--horizon hz`, where `hz` is a non-negative integer.

Your first job is to simulate a multi-armed bandit. You must read in the bandit instance and have a function to generate a random 0-1 reward with the corresponding probability when a particular arm is pulled. A single random seed will be passed to your program; you must seed the random number generator in your simulation with this seed. If any of your algorithms are randomised, they must also use the same seed for initialisation.

Given a seed and keeping other input parameters fixed, your entire experiment must be **deterministic**: it should execute the same way and produce the same result. Of course, the execution will be different for different random seeds; the point being made is that of repeatability for a given seed. You should be able to implement this property by initialising all the random number generators in your program based on the seed provided as input: you should not leave them unseeded or use strategies such as seeding based on system time. Make sure you understand this requirement; if the behaviour of your code does not get fixed by the input random seed (keeping other input parameters fixed), you will not receive any marks for the assignment.

Having set up the code to pull arms and generate rewards, you must implement the following sampling algorithms: (1) epsilon-greedy, (2) UCB, (3) KL-UCB, and (4) Thompson Sampling. You are free to make assumptions on unspecified aspects such as how the first few pulls get made, how ties get broken, how any algorithm-specific parameters are set, and so on. But you must list all such assumptions in your report. The two external parameters to the given set of algorithms are epsilon for epsilon-greedy sampling, and a threshold for Task 4, both of which will be passed from the command line. Recall that on every round, an algorithm can only use the sequence of pulls and rewards up to that round (or statistics derived from the sequence) to decide which arm to pull. Specifically, it is illegal for an algorithm to have access to the bandit instance itself (although `bandit.py` has such access).

Passed an instance, a random seed, an algorithm, epsilon, threshold, and a horizon, your code must run the algorithm on the instance for "horizon" number of pulls and note down the cumulative reward REW. Subtracting REW from the maximum expected cumulative reward possible (the product of the maximum mean and the horizon) will give you REG, the cumulative regret for the particular run. Note that this number can be negative (and might especially turn up so on small horizons—why?). When the algorithm terminates, `bandit.py` should output a **single** line with nine entries, separated by commas and terminated with a newline ('\n') character. The line must be in this format; `outputFormat.txt` contains a few such lines (in which REG and HIGHS are set to arbitrary values just for illustration).

> instance, algorithm, random seed, epsilon, scale, threshold, horizon, REG, HIGHS

The last entry, HIGHS, is specific to Task 4, and is explained below. However, you must print out a value (say 0) for all the tasks. Similarly, note that epsilon only needs to be used by `bandit.py` if the algorithm passed is `epsilon-greedy-t1`; for other algorithms, it is a dummy parameter. Your output must still contain epsilon (either the value passed to it or any other value) to retain the nine-column format. Your REG value for Task 4 is similarly a dummy; up to you whether to print out the actual regret or to put a placeholder value such as 0.

We will run your code on a subset of input parameters and validate the output with an automatic script. You will not receive any marks for the assignment if your code does not produce output in the format specified above.

Once you have finished coding `bandit.py`, run `check.sh` to make sure that you correctly read in all the command line parameters, and print the output as we described above. While testing your code, we will use a different version of `check.sh`—with different parameters—and call it inside your `submission` directory.

Task 2

This tasks considers the tuning of the exploration rate in UCB. In particular, suppose we take

ucb_arm = empirical_mean_arm + exploration_bonus,

where exploration_bonus is defined as the square root of

c × ln(total_pulls) / pulls_arm.

The default value of c is 2, but you can see that it can be larger or smaller to control the amount of exploration.

The aim of Task 2 is to optimise the "scale" c separately on the five instances provided for this task, and to interpret the results obtained. Generate the data described in the section below. Based on this data, write down in your report: (1) the value of c that gave the lowest regret for each of the five instances, and (2) what trend is observed across the instances, along with an explanation for it.

Task 3

In tasks 1 and 2, we assumed that the rewards from the bandit follow Bernoulli distributions. In this task and the next, we assume that the reward values are drawn from a distribution with a finite support that is fully contained in [0, 1]. This means that the reward obtained on pulling an arm can take more than two (but still one of a finite set of) values. The probability values associated with each of the individual values from the support can be different for different arms though, and are given in the bandit instance. Naturally, your `bandit.py` program has full access to the instance, but the internal sampling logic must only use the support (not access the associated probabilities) in addition to the history of pulls of each arm.

In Task 3, you are required to adapt your code from Task 1 so as to handle the larger support of the rewards. We will evaluate you based on the regret achieved by your algorithms: make your best effort to produce as low values as possible across a range of instances. While reporting regret, make sure you generalise your calculation from Task 1 correctly (that is, in the formula for REG).

It is a good idea to think about a principled approach to solve Task 3, rather than hash together different tricks for different instances/horizons. Describe your approach (and the justification) in your report: it will be a key aspect of our evaluation (over and above the cold numbers it yields).

Task 4

Task 4 also works on instances similar to the ones in Task 3: that is, ones in which the rewards come from a finite support contained in [0, 1]. In this task, our aim is to maximise the expected number of times we get rewards exceeding a threshold. We only care about the comparison between the reward and the threshold; it is immaterial what the reward value is so long as it exceeds the threshold. Concretely, call an outcome HIGH if it exceeds the threshold passed to the program; call it LOW otherwise (that it, it is less than or equal to the threshold). Your job is to write an algorithm to maximise the expected number of HIGH rewards (the total number of such rewards over the horizon is denoted HIGHS).

Once again, refashion your code from tasks 1 and 3 to provide an efficient sampling algorithm for this requirement. And once again, try to implement a conceptually-sound approach. Describe your approach in the report.

Output Data, Plots, Report

Having written all your code in `bandit.py` (which possibly calls other programs), now it's time to generate some data and evaluate your algorithms.

To generate data for Task 1, run `bandit.py` for every combination of

> **instance** from `"../instances/instances-task1/i-1.txt"`; `"../instances/instances-task1/i-2.txt"`; `"../instances/instances-task1/i-3.txt"`,
> **algorithm** from `epsilon-greedy-t1` with `epsilon` set to 0.02; `ucb-t1`, `kl-ucb-t1`, `thompson-sampling-t1`,
> **horizon** from 100; 400; 1600; 6400; 25600; 102400, and
> **random seed** from 0; 1; ...; 49.

To generate data for Task 2, run `bandit.py` for every combination of

> **instance** from `"../instances/instances-task2/i-1.txt"`; `"../instances/instances-task2/i-2.txt"`; `"../instances/instances-task2/i-3.txt"`, `"../instances/instances-task2/i-4.txt"`, `"../instances/instances-task2/i-5.txt"`,
> **algorithm** set to `ucb-t2`,
> **scale** from 0.02; 0.04; 0.06; ...; 0.3;
> **horizon** equal to 10000,
> **random seed** from 0; 1; ...; 49.

To generate data for Task 3, run `bandit.py` for every combination of

> **instance** from `"../instances/instances-task3/i-1.txt"`; `"../instances/instances-task3/i-2.txt"`,
> **algorithm** set to `alg-t3`,
> **horizon** from 100; 400; 1600; 6400; 25600; 102400, and
> **random seed** from 0; 1; ...; 49.

To generate data for Task 4, run `bandit.py` for every combination of

> **instance** from `"../instances/instances-task4/i-1.txt"`; `"../instances/instances-task4/i-2.txt"`,
> **algorithm** set to `alg-t4`,
> **threshold** from `0.2`; `0.6`,
> **horizon** from 100; 400; 1600; 6400; 25600; 102400, and
> **random seed** from 0; 1; ...; 49.

It is best that you write your own wrapper script for generating the output for all these input configurations. Place all the output lines in a file named `outputData.txt`. Notice that the total number of lines must be 9150 = 3600 (for Task 1) + 3750 (for Task 2) + 600 (for Task 3) + 1200 (for Task 4). It will take you at least 5 to 10 hours (depending on your system configuration) to generate data (especially for longer horizons), and so do not leave this task to the last minute. Since data for shorter horizons will anyway be generated as a part of the longer-horizon experiments, you might be able to save some time by recording intermediate regret values. However, your submitted `bandit.py` file must still only print a single line corresponding to the horizon passed to it.

You will generate a total of 8 plots.

- Task 1: Create one plot for each instance, with horizon on the x-axis (use a log scale) and regret on y-axis. Each plot will contain 4 lines: one for each algorithm.
- Task 2: Create a single plot with one line for each instance, with scale `c` on the x-axis, and regret on the y-axis.
- Task 3: Create one plot for each instance, with horizon on the x-axis (use a log scale) and regret on y-axis. Each plot will contain only a single line: that for your algorithm.
- Task 4: ~~Create one plot for each instance, with horizon on the x-axis (use a log scale) and HIGH on y-axis. Each plot will contain only a single line: that for your algorithm.~~ Although you report HIGHS in your raw output for this task, the more useful quantity to track would be the "regret" equivalent of HIGHS. Let MAX-HIGHS denote the maximum expected number of HIGH rewards possible for an instance; you should be able to calculate it from the instance file, the threshold, and the horizon. Let HIGHS-REGRET = MAX-HIGHS - HIGHS. In your plots for this task, the average value of HIGHS-REGRET over 50 runs is to be provided on the y-axis, with the horizon on the x-axis (on a log scale). You will create a separate plot for each threshold value for each instance: hence, 4 plots in total for this task.

In your plots, each point will give the average regret or ~~HIGHS~~ HIGHS-REGRET values from the 50 random runs at the particular horizon for the algorithm. Make sure you provide a clear key so the plot is easy to follow.

Include all ~~8 graphs~~ 10 graphs in a file called `report.pdf`, which should also state any assumptions in your implementation and provide your interpretations of the results. Feel free to put down any observations that struck you while working on this assignment. Do not leave your graphs as separate files: they must be embedded in `report.pdf`.

Submission

Place these items in the `submission` directory.

> `bandit.py` and all the code that it needs to run.
> `outputData.txt`
> `report.pdf`
> `references.txt` (see the section on Academic Honesty on the course web page)

Compress the directory into `submission.tar.gz` and upload it on Moodle under Programming Assignment 1. **Your submission will not be evaluated (you will get a score of 0) if the name or format of your compressed directory is different from** `submission.tar.gz`.

Evaluation

We will evaluate you based on your report, and also run your code to validate the results reported. If your code does not run on the cs747 docker container or your report is absent/incomplete, you will not receive any marks for this assignment.

6 marks are reserved for the correctness of your implementation in Task 1, along with the report containing the plots and the interpretation of the results. (1 mark each for epsilon-greedy and UCB, and 2 each for KL-UCB and Thompson Sampling.) Absence of the plots and implementation details of any algorithm will lead to zero marks for that particular algorithm.

2 marks are allotted to Task 2. We will evaluate you based on the graph you obtained and your reasoning in `report.pdf`.

2 marks each are allotted for tasks 3 and 4. We will evaluate the performance of your algorithms on the instances provided (and possibly other ones), and also verify the explanations/descriptions provided in the report. In each case, implement the best algorithm you are able to think of that can work well across a range of instances.

The TAs and instructor may look at your source code and notes to corroborate the results obtained by your agent, and may also call you to a face-to-face session to explain your code.

Deadline and Rules

Your submission is due by 11.55 p.m., Thursday, September 9, 2021. Finish working on your submission well in advance, keeping enough time to generate your data, compile the results, and upload to Moodle.

Your submission will not be evaluated (and will be given a score of zero) if it is not uploaded to Moodle by the deadline. Do not send your code to the instructor or TAs through any other channel. Requests to evaluate late submissions will not be entertained.

Your submission will receive a score of zero if your code does not execute on the cs747 docker container. To make sure you have uploaded the right version, download it and check after submitting (but well before the deadline, so you can handle any contingencies before the deadline lapses).

You are expected to comply with the rules laid out in the "Academic Honesty" section on the course web page, failing which you are liable to be reported for academic malpractice.