CS 747: Programming Assignment 1

Total marks: 12

(Prepared by Vedang Gupta)

This assignment tests your understanding of the regret minimisation algorithms discussed in class, and ability to extend them to different scenarios. There are 3 tasks, which add up to 12 marks. To begin, in Task 1, you will implement UCB, KL-UCB, and Thompson Sampling, more or less identical to the versions discussed in class. Task 2 involves maximising the reward for a bandit setting where arm pulls are randomly chosen from a query set and come at a cost as a function of the number of arms provided in the query set. Task 3 involves investigating the effect of the epsilon parameter in the epsilon-greedy algorithm.

Prerequisite Software

All the code you write for this assignment must be in Python 3.9.6 or greater. You can install Python 3.9.6 for your system from here.

Your code will be tested using a python virtual environment. To set up the virtual environment, follow the instructions provided in `virtual-env.txt` which is included in the compressed directory linked below.

Code Structure

This compressed directory has all the python files required. `bernoulli_bandit.py` defines the BernoulliBandit which, strictly speaking, you do not need to worry about. It is, however, advised that you read through the code to understand how the `pull` and other functions work. We have also provided `simulator.py` to run simulations and generate plots, which you'll have to submit as described later. Finally, there's `autograder.py` which evaluates your algorithms for a fixed few bandit instances, and outputs the score you would have received if we were evaluating your algorithms on these instances. The only files you need to edit are `task1.py`, `task2.py`, and `task3.py`. Do not edit any other files. You are allowed to comment/uncomment the final few lines of `simulator.py` which you can use to generate the plots for tasks 1 and 2.

For evaluation, we will use another set of bandit instances in the autograder, and use its score as is for approximately 75% of the evaluation. So if your code produces an error, it will directly receive a 0 score in this part. It will also get 0 marks if for any testcase whatsoever, the autograder takes over 5 minutes to run the testcase. The remaining part of the evaluation will be

done based on your report, which includes plots, and explanation of your algorithms. See the exact details below.

For tasks 1 and 2, you can expect that the number of arms in the bandit instances used in our undisclosed test cases will be at most 40, and similarly the horizon at most 20,000. (Note that longer horizons are used in your plots, which might take some time to generate).

To test your implementation against the given testcases, run the autograder as follows: `python3 autograder.py --task TASK`. Here `TASK` can be one of: `1, 2` or `all`. If `TASK` is `1`, then you also need to provide another argument: `--algo ALGO`, where `ALGO` can be one of: `ucb, kl_ucb, thompson`, or `all`.

Your code will be evaluated using the python virtual environment. Testing your code using the activated python virtual environment on your machine should be sufficient for your code to work during evaluation.

During evaluation we will be running your submitted files with our autograder framework which is a superset of the one we have provided you. Hence, you must make sure that your submitted files are compatible with the original framework.

Problem statements for tasks

Task 1 (4 marks)

In this first task, you will implement the sampling algorithms: (1) UCB, (2) KL-UCB, and (3) Thompson Sampling. This task is straightforward based on the class lectures. The instructions below tell you about the code you are expected to write.

Read `task1.py`. It contains a sample implementation of epsilon-greedy for you to understand the `Algorithm` class. You have to edit the `__init__` function to create any state variables your algorithm needs, and implement `give_pull` and `get_reward`. `give_pull` is called whenever it's the algorithm's decision to pick an arm and it must return the index of the arm your algorithm wishes to pull (lying in 0, 1, ... self.num_arms-1). `get_reward` is called whenever the environment wants to give feedback to the algorithm. It will be provided the `arm_index` of the arm and the `reward` seen (0/1). Note that the `arm_index` will be the same as the one returned by the `give_pull` function called earlier. For more clarity, refer to `single_sim` function in `simulator.py`.

Once done with the implementations, you can run `simulator.py` to see the regrets over different horizons. Save the generated plot and add it your report, with apt captioning. You may also run `autograder.py` to evaluate your algorithms on the provided test instances.

Task 2 (6 marks)

This task involves dealing with a bandit instance where pulling arms has an associated cost. In this setting, you can request pulls from an oracle by providing a query set (S) which is an array of arm indices. The oracle then chooses an arm from this set uniformly at random and provides you with both the reward obtained and the arm it decided to pull. For this action, the oracle charges you a cost of 1/|S|. For example, you might provide the query set S = {0, 1, 4}. The oracle then chooses one of these arms with probability 1/3: say arm 4. Then it obtains a reward 0 and returns the tuple (4, 0) and incurs a cost of 1/3. Your net reward for this pull is then -1/3. Your task is to come up with a good algorithm to maximise the expected net reward = (total bandit reward - total oracle cost) for this costly bandit setting.

Similar to `bernoulli_bandit.py`, `set_bandit.py` implements this costly bandit setting. For the above task, you must edit the file `task2.py`. The structure of the algorithm class is very similar to that in `task1.py`. Instead of a single arm index, you must return the query set as an array in the `give_query_set` function even if you only provide a query set of one arm. You must specify the `get_reward` function appropriately assuming you get the same inputs: `arm_index` and `reward`. Note that the `reward` input to the function does not contain the cost incurred. You can receive the total net reward through the `task2()` function in `simulator.py`.

Task 3 (2 marks)

In this task you will investigate the effects of the epsilon parameter in the epsilon-greedy algorithm. Take the bandit instance with arm means: [0.7, 0.6, 0.5, 0.4, 0.3] and fix the horizon to be 30000. Plot a graph showing the change in regret as you vary epsilon from 0 to 1 (both inclusive) in steps of 0.01. Your algorithm implementation should be such that each arm is pulled once at the start so that the empirical means are well-defined. You may tiebreak as you wish but make sure to describe your tiebreaking rule in the report. The regret should be calculated as an average over atleast 50 runs. Clearly state your observations from the plot, and explain the reason(s) behind the observations. What is the value of epsilon for which regret is lowest?

For this task, you must edit the file `task3.py`. Look at code from `task1.py` and `simulator.py` and use it to help write code for `task3.py`. First, write an algorithm class that implements the epsilon-greedy algorithm. To create the plot, add appropriate helper simulation functions: `single_sim_task3()`, `simulate_task3()` and `task3()` similar to the ones defined for the first two tasks in `simulator.py`.

Report

Your report needs to have all the plots (UCB, KL-UCB, Thompson) that `simulator.py` generates (uncomment the final lines) as well as your plots for task 3. There are 4 plots in total (3 for task 1, 1 for task 3). You may, of course, include any additional plots you generate. Your plots should be neatly labelled and captioned for the report. For Task 3, as explained in the questions above, state your observations and explain the reasons behind them. In addition, you

need to explain your method for tasks 1 and 2. For task 1, explain your code for the three algorithms, and for task 2 you must give a reasonable explanation for your approach to the problem.

Submission

You have to submit one tar.gz file with the name `(your_roll_number).tar.gz`. Upon extracting, it must produce a folder with your roll number as its name. It must contain a `report.pdf` - the report as explained above, and three code files: `task1.py, task2.py, task3.py`. You must also include a `references.txt` file if you have referred to any resources while working on this assignment (see the section on Academic Honesty on the course web page).

Evaluation

The assignment is worth 12 marks. For Task 1, 3 marks are for the code and will be evaluated by the autograder, and 1 mark is for the report. For Task 2, 5 marks are for the code and will be evaluated by the autograder, and 1 mark is for the report and the explanation of your algorithm.

For Task 1 and 2, we will use the following marking system: "FAILED" testcases contribute 0 marks, and "PASSED" testcases contribute 1 mark, the total of which will then be normalized to the task's assigned marks depending on the number of testcases. The visible testcases will contribute to 25% of the marks and the rest 75% will come from the hidden testcases that we will use once you have submitted the assignment.

Task 3 is worth 2 marks. The evaluation is based solely off your plots, the plotting script used, and your explanation of the results generated. There will be no autograded testcases for task 3.

The autograder used for evaluating will use a different set of test instances from the ones provided. Note that runtime errors in the script will lead to 0 marks for that test instance. You will also be given 0 marks for a test instance if your code takes more than 5 minutes to run for a test instance.

Deadline and Rules

Your submission is due by **11.59 p.m., Tuesday, February 18, 2025**. Finish working on your submission well in advance, keeping enough time to test your code, generate your plots, 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 using the given python virtual environment. 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.