CS 747: Programming Assignment 1

Total marks: 15

(Prepared by Vedang Gupta and Jujhaar Singh)

This assignment tests your understanding of the regret minimisation algorithms discussed in class, and ability to extend them to different scenarios. There are 4 tasks, which add up to 15 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 2A involves studying the effect of the difference between the means of two arms on the regret accumulated by the UCB algorithm. Task 2B involves studying and comparing the effect of the value of the means on the regret accumulated by UCB and KL-UCB while keeping the difference between the means constant. Task 3 involves maximising the reward for a bandit setting where pulls are noisy and give faulty outputs with a certain probability. Task 4 involves dealing with multiple bandit instances where the bandit instance for a particular pull is chosen at random. Your task would be to maximise the reward for this multiple-bandit setting.

Pre-requisite Software

All the code you write for this assignment must be in Python 3.8.10. You can install Python 3.8.10 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 code and data files for the assignment. `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 `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`, `task3.py`, and `task4.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 Task 1 or experiment with tasks 3 and 4.

For evaluation, we will use another set of bandit instances in the autograder, and use its score 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 task whatsoever, the autograder takes over 20 minutes to run the task. 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, 3 and 4, 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 test cases, run the autograder as follows: `python3 autograder.py --task TASK`. Here `TASK` can be one of: `1, 3, 4` 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. Ensuring that your code using the activated python virtual environment passes the autograder tests on your machine should be sufficient for your code to work during evaluation.

Problem Statements for Tasks

Task 1

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 is 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 about 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

Part A

This task explores the effect of the difference between the means of arms on the regret accumulated by the UCB algorithm. For this task, take two-armed bandit instances (with means [p1, p2]) with the higher mean arm fixed (say, p1), and vary the other arm's mean (p2) from 0 to p1. For the assignment, p1 = 0.9, and p2 varies from 0 to 0.9 (both inclusive) in steps of 0.05. Do this for a horizon of 30000. Plot the variation of regret with p2. Clearly state your observations from the plot, and explain the reason(s) behind the observations.

Part B

This task involves comparing the behaviour of UCB and KL-UCB algorithms on two-armed bandits. The task is to compare the effect of the value of the means on the algorithms while keeping the difference between the means fixed. For this task, take Delta = p1 - p2 = 0.1. Again, vary p2 from 0 to 0.9 (both inclusive) in steps of 0.05 for a horizon of 30000. Plot the variation of regret for both algorithms on different plots with respect to the instances (you may plot with respect to either p1 or p2, but clearly label that in your plots). State and compare your observations for the variations of regret for each algorithm and explain the reason behind your observations.

For both of the above tasks, you can edit any code in the file `task2.py`. It includes all the necessary classes and functions to simulate a bandit instance but is well isolated from all the other tasks, so you are free to edit them as you wish. You only need to call the function `task2` to generate your array of average regrets given the two arrays of p1s and p2s. The bandit instances are created according to corresponding indexes of the arrays. For example, if your array for p1s = [0.2, 0.3] and p2s = [0.1, 0.2], the function will return an array of regrets corresponding to the instances (0.2, 0.1) and (0.3, 0.2).

Task 3

This task involves dealing with a bandit instance where your pulls are no longer guaranteed to be successful and have a probability of giving faulty outputs. When you pull an arm, it has a certain known probability of giving the correct output of the arm, otherwise it returns a 0 or 1 uniformly at random. Your task is to come up with a good algorithm to maximise the reward for this faulty bandit setting.

For the above task, you must edit the file `task3.py`. The structure of the algorithm class is very similar to that in `task1.py` with the only difference being here you are also provided with the probability of a faulty pull labelled `fault` in the code. Again, you must specify the `give_pull` and `get_reward` functions appropriately.

Task 4

This task involves dealing with two bandit instances at once! In this task, whenever you specify an arm index to pull, one of two given bandit instances is chosen uniformly at random, and the arm corresponding to your provided index is pulled (both instances have equal number of arms). Once an arm is pulled, the environment returns the reward obtained along with which bandit instance was chosen for that pull. Your task is to come up with a good algorithm to maximise the reward for this multi-multi-armed bandit setting.

For the above task, you must edit the file `task4.py`. The structure of the algorithm class is very similar to that in `task1.py` with the only difference being here the `get_reward` function also takes as input `set_pulled` which specifies which bandit instance was chosen (0 = 1st instance, 1 = 2nd instance). Again, you must specify the `give_pull` and `get_reward` functions appropriately.

Report

Your report needs to have all the plots that `simulator.py` generates as well as your plots from `task2.py`. There are 6 plots in total (3 for Task 1, 1 for Task 2A, 2 for Task 2B). You do not need to include the epsilon-greedy plot in your report. You may, of course, include any additional plots you generate. Your plots should be neatly labelled and captioned for the report. For Task 2, 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, 3 and 4. For Task 1, explain your code for the three algorithms (implementational details, parameter settings, etc.), and for tasks 3 and 4, give a clear description of your approaches to tackle the problems. If your descriptions are not sufficiently clear and informative, you will not receive full marks.

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 four code files: `task1.py, task2.py, task3.py, task4.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 15 marks. For Task 1, 4 marks are for the code and will be evaluated by the autograder, and 1 mark is for the report. For Task 2, each part (A and B) are worth 2 marks. The evalaution 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 2.

For tasks 3 and 4, 2 marks are for the code and will be evaluated by the autograder, and 1 mark is for the report. For tasks 3 and 4, we will use a partial marking system, where "FAILED" test cases contribute 0 marks, "PARTIALLY PASSED" test cases contribute 0.5 marks, and "PASSED" testcases contribute 1 mark, the total of which will then be normalized to 2 marks depending on the number of test cases. "PARTIALLY PASSED" is decided according to a softer threshold on the reward accumulated, while "PASSED" is decided according to a more competitive threshold.

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 20 minutes to run for that test instance.

Deadline and Rules

Your submission is due by **11.55 p.m., Sunday, September 10**. 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.