CS 747: Programming Assignment 1

Total marks: 15

(Prepared by Vibhav Aggarwal and Sheel Shah)

This assignment tests your understanding of the regret minimisation algorithms discussed in class, and ability to extend them to different scenarios. There are three tasks, each worth 5 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 coming up with an algorithm for *batched* sampling. The idea is that at every decision making step, the algorithm must specify an entire batch of arms to pull (for example, if the batch size is 100, it could be split as perhaps 25 pulls for arm 1, 55 pulls for arm 2, and 20 pulls for arm 3). All these pulls are performed and the results returned to the algorithm in aggregate before its next batch of pulls. In Task 3, you need to come up with an algorithm for the case when the horizon is *equal* to the number of arms, but it is given that the arm means are distributed uniformly (so if the horizon is 100, the arm means are a permutation of [0, 0.01, 0.02, ..., 0.99]). The theory developed in class applies meaningfully only when the horizon is large; how would you deal with shorter horizons? The fact that the bandit instance comes from a restricted family could possibly help.

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, and are already imported in the files you need to complete.

Code Structure

This compressed directory has all the 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 `batch_pull` 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`. It is strongly recommended that in any code that you write, which involves generating random numbers, you fix the seed for your generation process. The consequence is that your code will produce identical results each time (so long as we call it from an outer loop that also has its random seed fixed). This is a good practice while running experiments with programs.

For evaluation, we will use another set of bandit instances in the autograder, and use their scores as is for 80% of the evaluation. For a particular test instance, the pass/fail criterion of the autograder is determined based on your regret lying within 1.5 times of the regret of our reference implementation. If your code produces an error, it will directly receive a 0 score in the autograded tasks. It will also get 0 marks if for any subtask whatsoever, the autograder takes over 20 minutes to run the subtask. 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.

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 to the algorithm: your code can use this feedback to update the data structures maintained by your agent. 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 to your report, with suitable commentary (ideally 4-5 lines describing what you see, what issues you faced, any surprising patterns). You may also run `autograder.py` to evaluate your algorithms on the provided test instances.

Task 2

We describe the problem statement that was briefly discussed completely now. The algorithm is given the number of arms of the bernoulli bandit `num_arms`, a `horizon` and a `batch_size`. The `give_pull` function will be called `horizon/batch_size` times (which can be assumed to be an integer). In every call, it must return the next `batch_size` number of pulls the algorithm wishes to make. The function must do so in a specific format: it has to return two lists, one containing the arm indices that it wishes to pull, and the other containing the number of times each of those indices must be pulled. For example, in a 10-armed bandit instance with batch_size 20, a possible return of

the `give_pull` function could be ([2, 4, 9], [10, 4, 6]). Note that your function should generalize to arbitrary batch_sizes, as long as the given batch_size is a factor of the horizon (the batch_size could be just 1, or it could also be of the order of the horizon). The autograder/simulator will proceed and pull these arms according to their respective counts, and then provide feedback to the `get_reward` function. The feedback is provided as a dict, where the keys are the arm indices, and the rewards are a numpy array of 0s and 1s that were seen. So a possible input (`arm_rewards`) to the `get_reward` function for the above batch pull could be {2: np.array([1, 1, 1, 0, 1, 1, 0, 1, 0, 1]), 4: np.array([1, 1, 0, 0]), 9: np.array([0, 1, 0, 1, 0, 0])}. Again, you can read `single_batch_sim` for more clarity. The regret is calculated over all the pulls over the horizon.

Once done with your implementation, you can run `simulator.py` to see the regrets for a fixed horizon over different batch sizes. Save the generated plot and add it to your report, with apt captioning. Take 4-5 lines (or more) to explain the trend you see, and justify your choice of the distribution of pulls in a given `batch_size`. You may also run `autograder.py` to evaluate your algorithms on the provided test instances.

Task 3

This task involves dealing with a bandit instance where the horizon is equal to the number of arms. So, for example, if there are 100 arms, then you are only allowed to pull 100 times. However, you are given that the arm means are distributed regularly (in arithmetic progression) between 0 and (1 - 1/numArms).

You need to come up with an algorithm to handle this situation effectively: can you do better than sampling each arm once? Implement your algorithm by editing the `task3.py` file. The APIs you need to implement are essentially the same as `task1.py`.

Once again, you can use `simulator.py` to see regrets as a function of horizon. You may also run `autograder.py` to evaluate your algorithm. Note that even if your algorithm passes the autograder tests, it might fail on the undisclosed tests that are used for evaluation. So you must not hardcode your method to make it work for only the given test instances. For this task, you will again plot regret against horizon (which is the same as the number of arms).

Report

Your report needs to have all the plots that `simulator.py` generates. There are 5 plots in total (3 for task 1 and 1 each for tasks 2 and 3). You do not need to include the epsilon-greedy plot in your report. In addition, you need to explain your method for each task. For task 1, explain your code for the three algorithms, and for tasks 2 and 3 explain your approach to the problem and provide justification to the trend seen in the plots.

Submission

You have to submit one tar.gz file with the name (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 15 marks. For every task, 4 marks are for the code and will be evaluated by the autograder, and 1 mark is for the report. 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 a test instance.

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 30, and similarly the horizon at most 10,000. (Note that longer horizons are used in your plots, which might take some time to generate). For Task 2, the batch size could be any legal number (that is, which perfectly divides the horizon) between 1 and the horizon. In Task 3, the horizon (equal to the number of arms) will be at most 10,000.

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`, `3`, 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`.

Deadline and Rules

Your submission is due by 11.55 p.m., Saturday, September 10. Finish working on your submission well in advance, keeping enough time to test your code, 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.