Multi-Armed Bandit

Vansh Gupta
49003814

Written report 1

For the MAB project, we were supposed to do the following:

1. Give a proper reasoning that why your environment is suitable for multi-armed bandit.
2. Design your own reward and return settings.
   Write codes for MAB algorithms and environments. (**bandit.py**)
3. Show more than two efforts to improve the inference accuracy.
   The efforts could include:
    - Employing several MAB models (Epsilon-greedy, UCB, ...)
    - Optimizing rewards
4. Write a report that contains 1-3.

And we were supposed to follow the following rules:

- Create your own arbitrary environment for a **single** agent.
- There should be more than **five possible actions** among each state.
- Use a win/lose (or termination) rule.
  Every win or lose should be determined right after an executed action (example: rock-scissors-paper game.)
- Design the opponent or environment reasonably (the environment could act simply but it must not be foolish).

For my game, I came to realize that there is not much that can be done with MAB given the requirement of game being stateless, and thus decided to do something fun instead. It should be noted that I asked about this topic well before it was brought up in one of the lectures. The topic I chose was speed dating. More specifically, the algorithm, with different policies, will slightly/ strongly nudge a person putting effort into pursuing a partner. I believe this is suitable because the gambler (person) has initially no knowledge about the machine (partner), there are >= 5 choices at every state and there is a good scope of trade-off between exploitation and exploration.

One thing to note is that I wrote my bandit in agents.py and not bandit.py as bandit.py was nowhere imported/referenced in the codebase (or at least the version that I seem to have)

For the dataset, I used the following publicly available dataset: https://www.kaggle.com/datasets/somesh24/speeddating

This data was gathered from participants in experimental speed dating events from 2002-2004. During the events, the attendees would have a four-minute "first date" with every other participant of the opposite sex. At the end of their four minutes, participants were asked if they would like to see their date again

The dataset also includes questionnaire data gathered from participants at different points in the process. With a feature set of 100+ and 8000+ number of rows, I knew I could transform it to fit the template code and change the template code a bit to get what I wanted.

First thing I did was use 17 attributes that a person judges themselves on, and used that as a unique identifier for users. The probability of 2 people having everything same was 1 in 10^{17}. Once the grouping was done, I did some data analysis, to find the following:

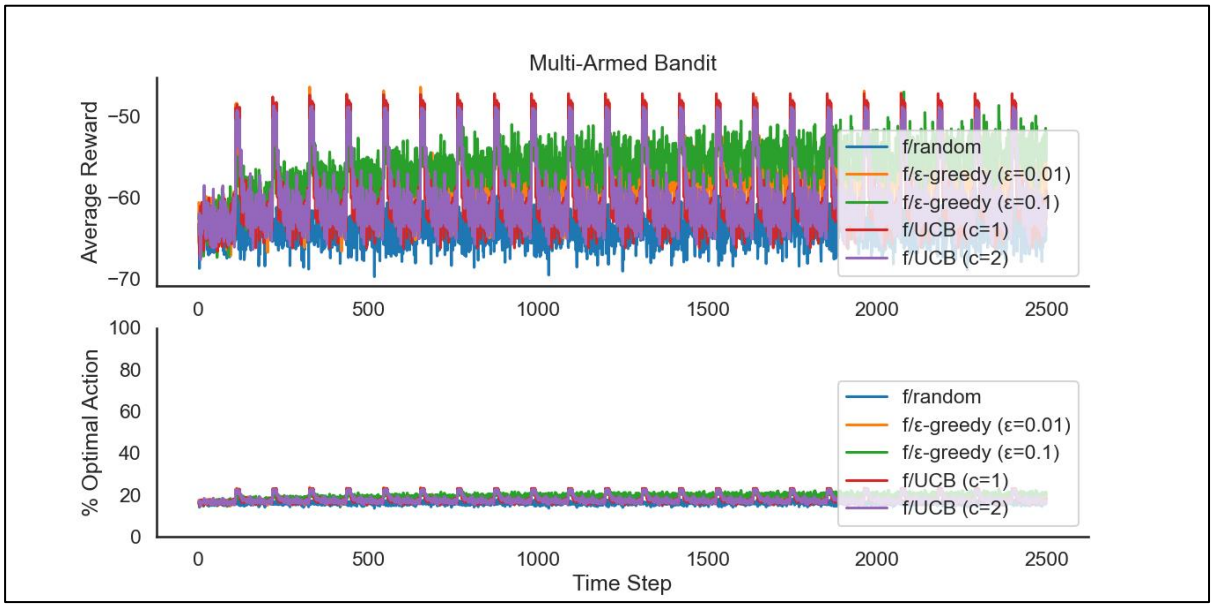
Number of dates	Number of users with that many dates
5	9
6	27
7	6
8	6
9	28
10	98
14	38
15	30
16	50
18	74
19	34
20	58
21	42
22	44
79	1

So given the instructions, I could use all of these. Next, I sort them based on some metric to fit the way the MAB algorithm works. Similar had been done in the horses code where they were sorted by the winrate for each race.

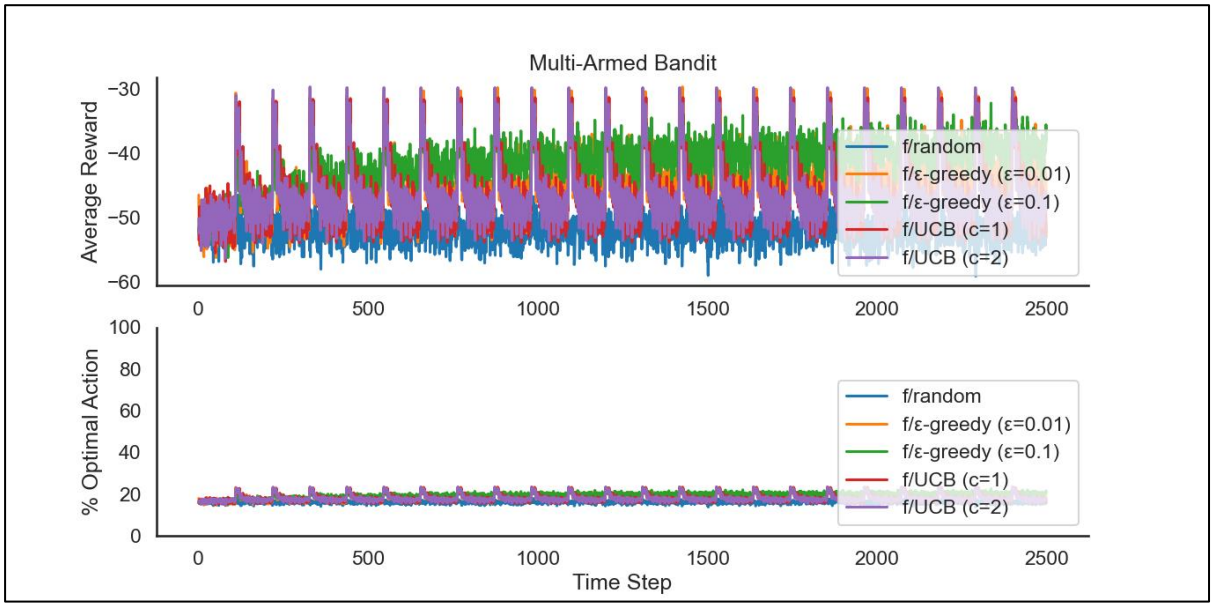
For this, I do a secondary data analysis, to find out the parameters which had more weight in getting a match. I found correlation of every feature with the match column to get to this result. I split them in 3 different types of sorting methods based on intuition and made sure these were different from the way reward was decided (Note: In rules, we were asked only to use win/lose rule, but I still made it fair for the other mode)

After debugging my code for a few days, the results that I got are as follows:

For each, the policies used are mentioned in the graph itself, and the associated reward is mentioned right below it:



(Dividend)

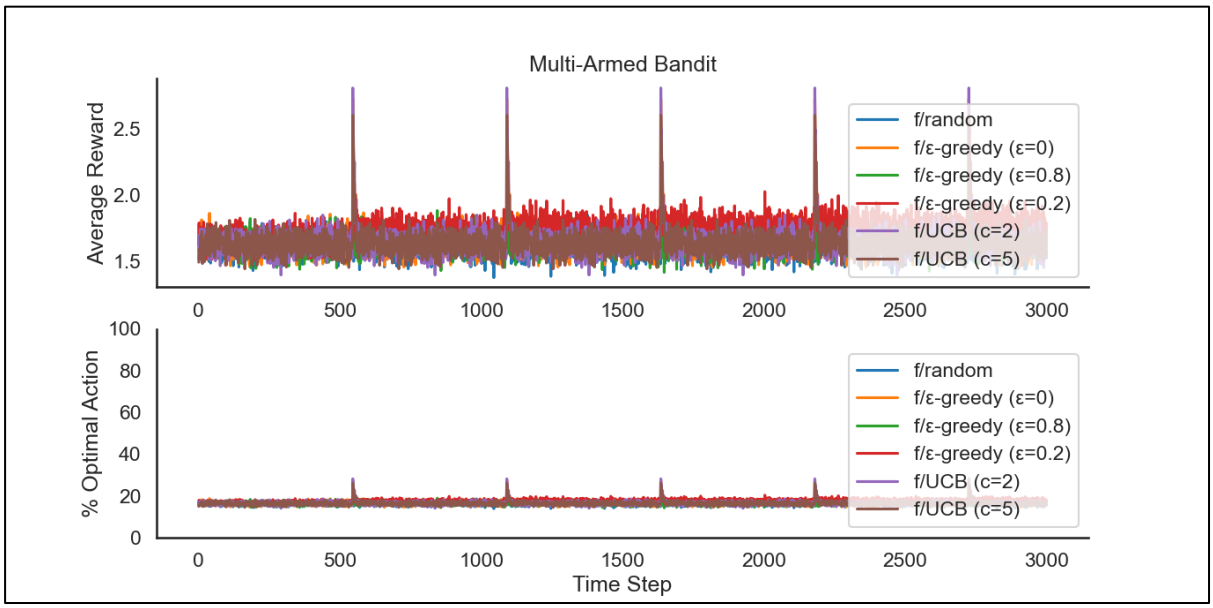


(Win-Loss)

#trials: 2500, #expts: 2500

(200, True) if self.mode == "winloss" else (20*min(race[action]["like"], race[action]["guess_prob_liked"]), True)

(-100, False)

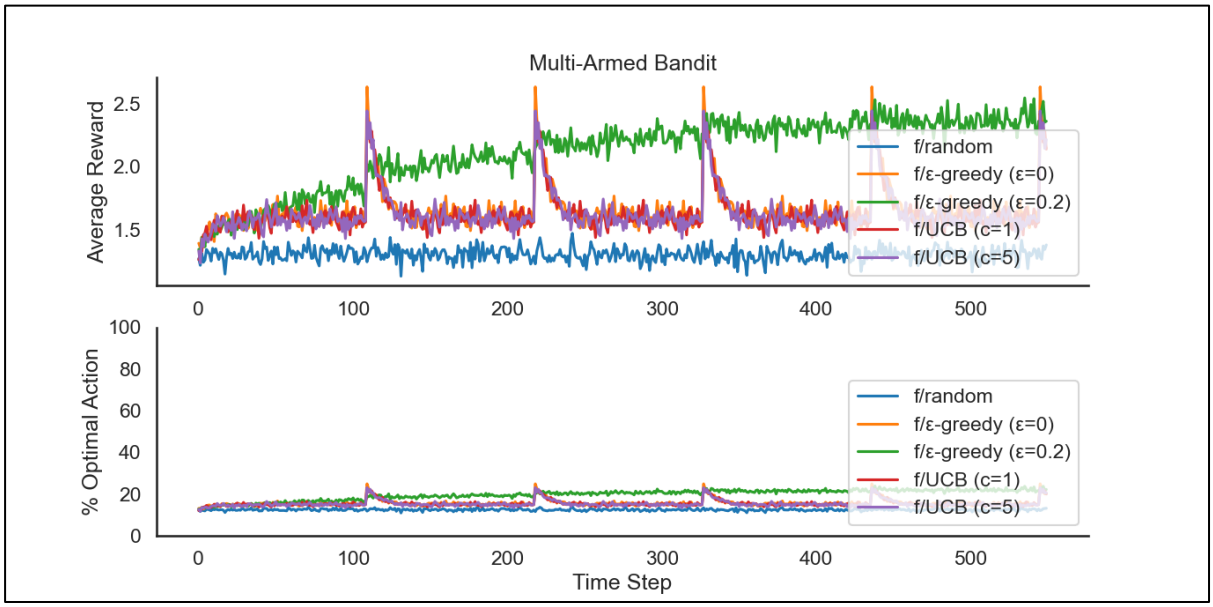


(Win-Loss)

#trials: 3000, #expts: 3000

(10, True) if self.mode == "winloss" else (1.7*min(race[action]["like"], race[action]["guess_prob_liked"]), True) # win

(0, False)



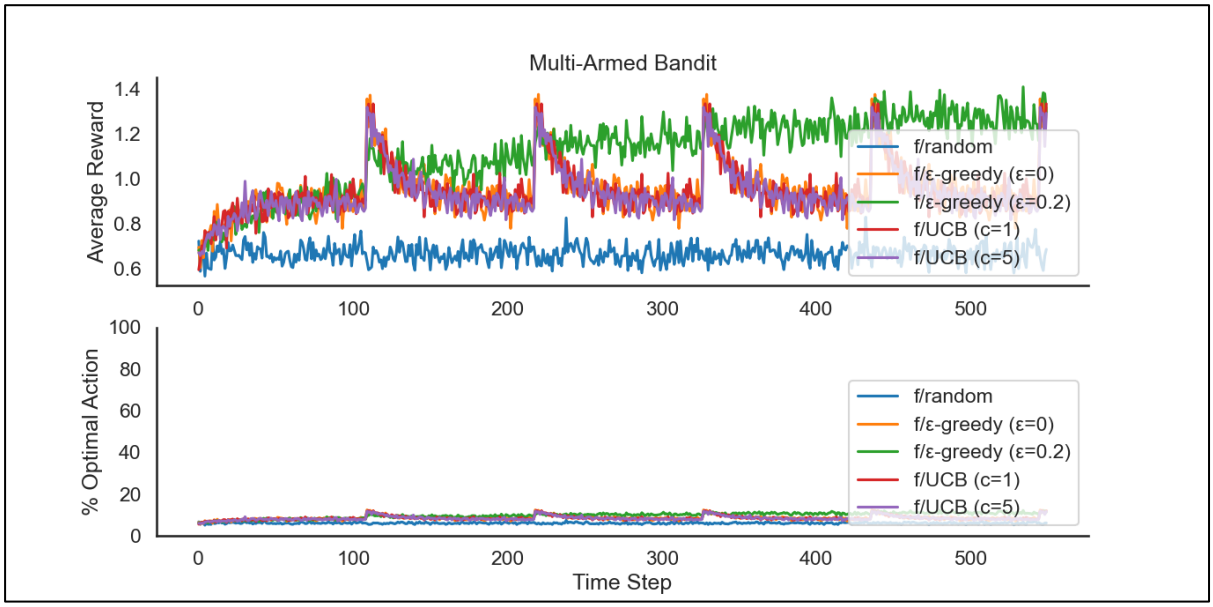
(Dividend)

#trials: 550, #expts: 5000

df1['sortType1'] = df1['interests_correlate']*(df1['attractive_partner'] + df1['intelligence_partner'] + df1['funny_partner'])

return (10, True) if self.mode == "winloss" else (1.7*min(race[action]["like"], race[action]["guess_prob_liked"]), True) # win

return (0, False)



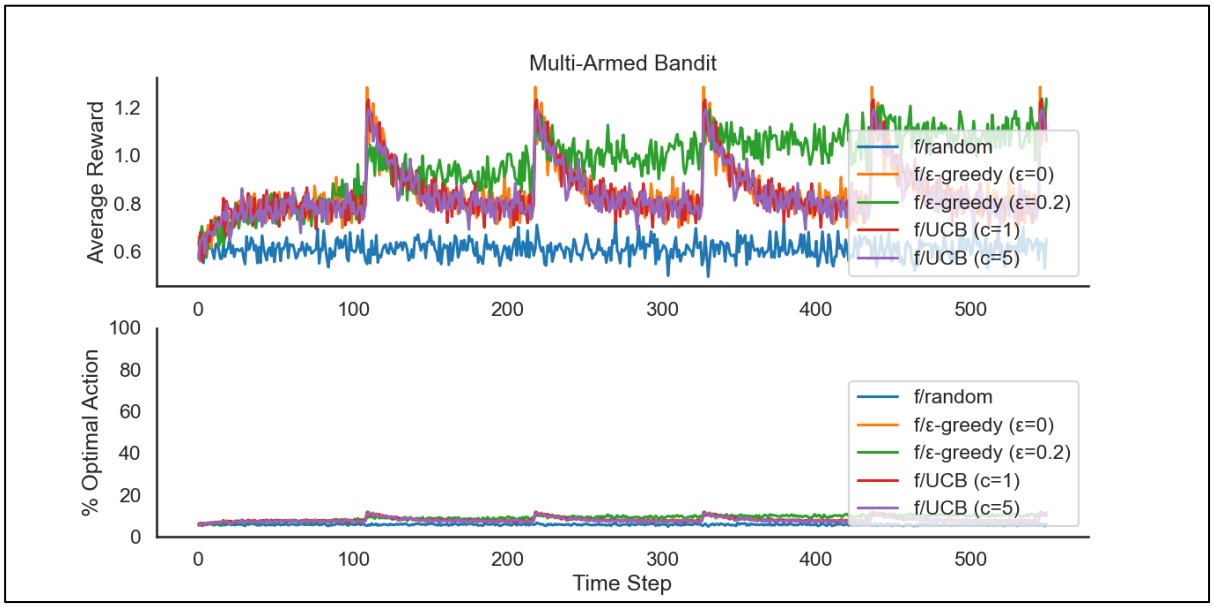
(Dividend)

#trials: 550, #expts: 5000

df1['sortType2'] = df1['interests_correlate']*df1[['like', 'guess_prob_liked']].min(axis=1)

return (10, True) if self.mode == "winloss" else (1.7*min(race[action]["like"], race[action]["guess_prob_liked"]), True) # win

return (0, False)



(Dividend)

#trials: 550, #expts: 5000

df1['sortType1'] = df1['interests_correlate']*(df1[['attractive_partner', 'attractive_o']].min(axis=1) + df1[['intelligence_partner', 'intelligence_o']].min(axis=1) + df1[['funny_partner', 'funny_o']].min(axis=1))

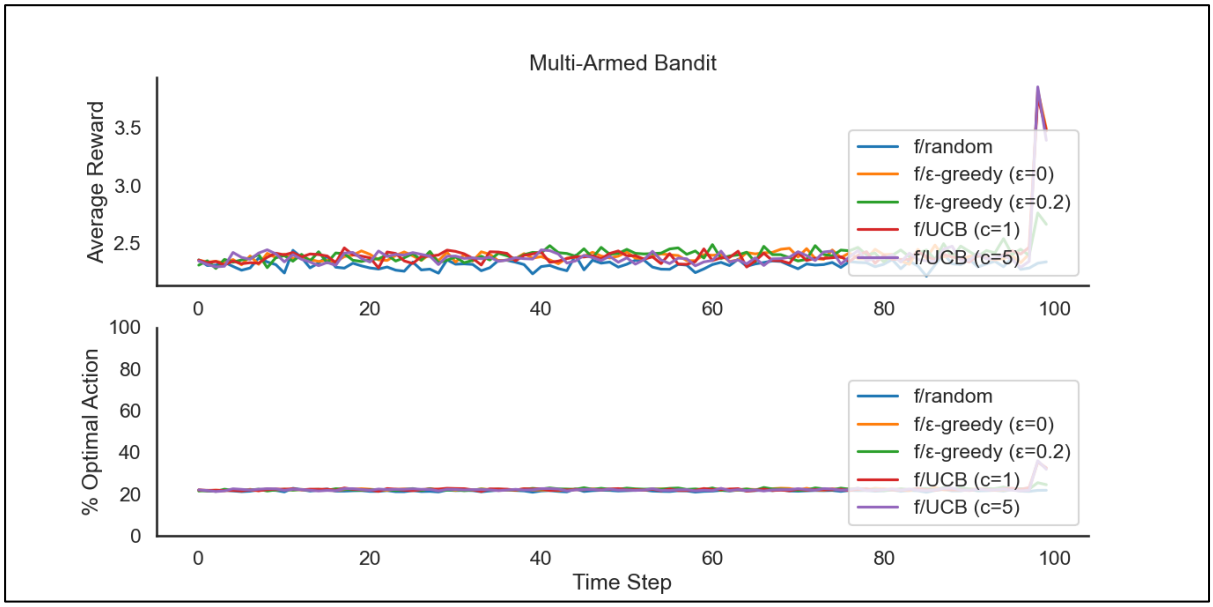
return (10, True) if self.mode == "winloss" else (1.7*min(race[action]["like"], race[action]["guess_prob_liked"]), True) # win

return (0, False)

From my various graphs, I tried to improve my hyperparameters as I went along. First, on my initial runs of the experiment, my reward always came out to be 0. Which is when I found out that due to try and exception block, I was running into an error and always returning 0 as the reward.

Then, I see a clear correlation between increase in the number of trials or experiments and the convergence of reward. However, it has to be noted that even though the reward is converging, the % of times optimal action was taken is still very low compared to the sample that we saw in the class. The reason for this can be that the sample had 3000 instances of data and that data was coherent in the sense that every race had 12 horses. So, while selecting a horse, the algorithm had definite choices. In my case, due to unavailability of a larger dataset, I had only 545 training instances and that too of varied length. So the algorithm could guess on which side of the spectrum better machines (partners) exist, but not learn the exact numbers.

However, running the model on just 98 instances of 10 partners per user gave the following results:



Here, we find a bit comparable results to what I saw earlier in terms of absolute values (and not the curve). This indicates that given a larger dataset, the algorithm would give better optimality

Next, I also change my reward values to see if it helps in achieving a better optimality, but it does not make that much of a difference

I could not change the condition of reward, as it was the best and most definite way to define the reward

After that, I take different sorting methods (3) to see which one is easier for the models to learn, and found out that the better one was where I only took partner's rating into account and not the person's own judgement.

Further, I tried multiple models to see which one understands the game better and found remarkably better results with the epsilon greedy method. Finally, I do some sweep over 5 different logspaced values of c between 0.001 and 1 (keeping everything else same as the one with best results) to understand the hyperparameter better. Here are the results for that: