Interactive Learning Homework 1

Soleiman BashirGanji
810103077

Solution Structure

Solution consists of 3 files. A `.ipynb` notebook, a `.py` script and a report (this file).
The `simulation.py` file contains the main body of the algorithms for all the three questions. It handles agents, algorithms, reward functions, etc.
On the other hand, the `hw1.ipynb` is where these codes are imported and used. For the sake of clarity, the notebook only contains different runs of algorithms and the visualizations/analysis related to them.

Q1

There are two classes for this problem in the `simulation.py`: `FoodRecommender` and `Environment`
This is how number of arms and probability of rewards are calculated:

```python
def __init__(self, student_id: str = '810103077'):
    self.n_arms = (int(student_id[-3:]) % 5) + 2
    self.p_action_rewards = [int(i) / 10 if int(i) else 0.5 for i in student_id[-self.n_arms:]]
```

FoodRecommender is the agent, and it has a function for performing an action and returning the resulting reward.
The `Environment` class has three functions, one for performing Epsilon-greedy, one for performing Thompson Sampling and one for performing UCB.

In the notebook, there is a function called `visualize_results()`. This function takes in several parameters and plots the rewards, regrets, action selections, mean and confidence intervals of the rewards and actions. This function is used to show the performance of the algorithms across the notebook.
I will explain the plots and results of this function for one of the instances in detail and will leave the rest with less details since they're already available in the notebook.

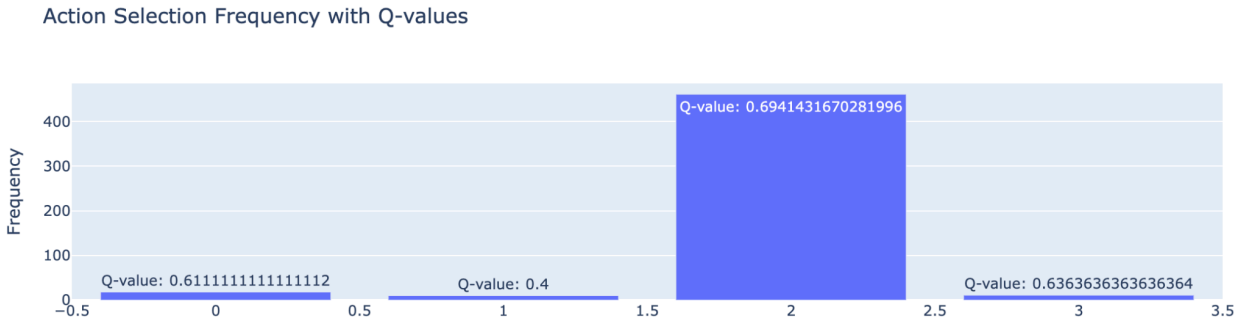
Epsilon-Greedy
I implemented a linear decay for the epsilon:

```python
current_epsilon = max(initial_epsilon - (decay_rate * iteration), 0)
```

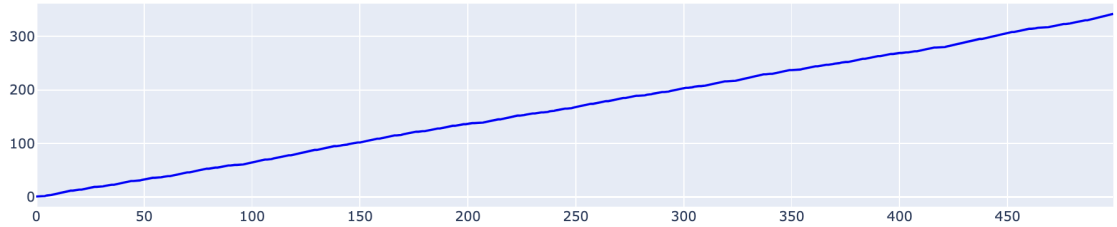
And ran it with different initiations. Here are the results of my first run:

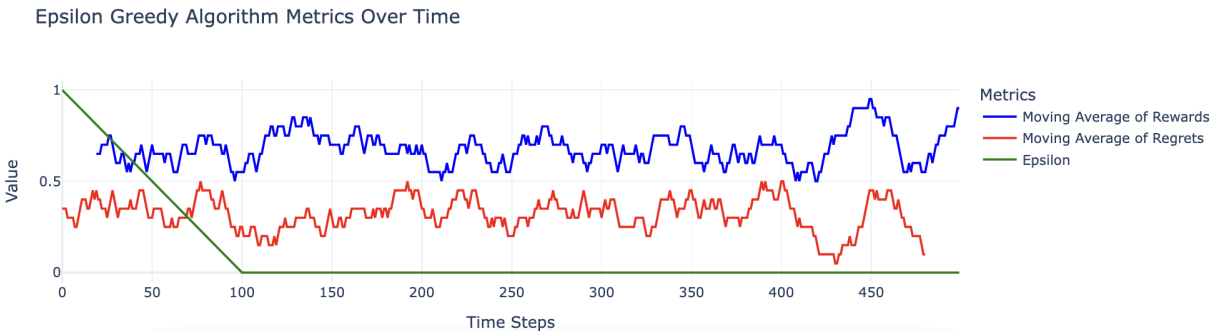
```
Mean Reward: 0.684, 95% CI: [0.6431093455633963, 0.7248906544366038]
Mean Regret: 0.316, 95% CI: [0.2751093455633963, 0.3568906544366037]
```

How many times each action was selected?



We can see that the algorithm correctly identified the optimal action.
Cumulative reward + Moving average of reward and regret:



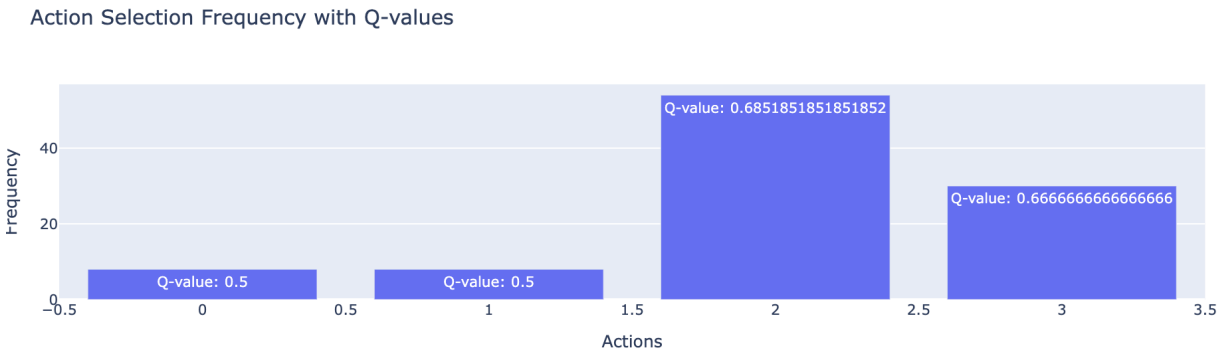


Same plots and analysis are in the notebook for other runs of this algorithm. My overall conclusion is that the algorithm finds the optimal action very quickly, and there's no need for long explorations.
The average reward is close to 0.7, this makes sense due to the Law of Large Numbers since the probability of reward for the optimal action is 0.7.

Thompson Sampling
The sampling is done like this:

```
samples = [np.random.beta(a=a, b=b) for a, b in zip(alphas, betas)]
```

If we get a reward, we increase alpha by one otherwise we increase beta. The rest is just performing actions and gathering information. There are several runs of the algorithm in the notebook. Similar to epsilon greedy, we can see that the policy quickly converges to the optimal action.



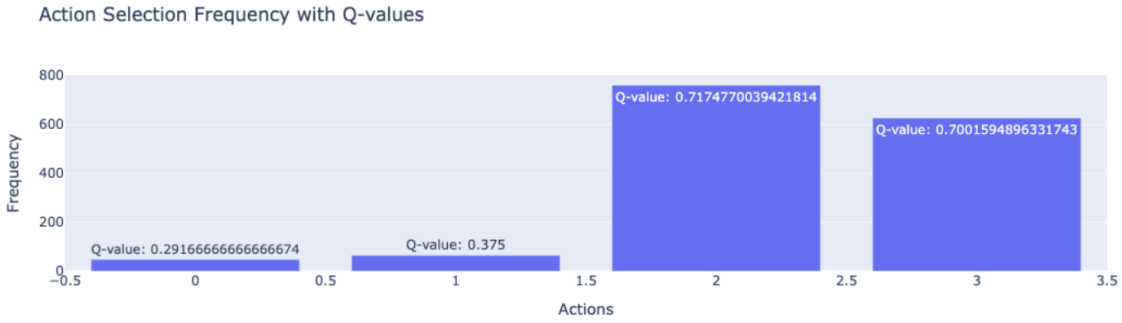
We can see that Thompson Sampling does not just try one optimal action. We have two optimal actions and the algorithm is able to identify both of them, but epsilon greedy would be biased towards only one of the two optimal actions.
This means that Thompson Sampling does not act greedy, and leaves more space for exploration.

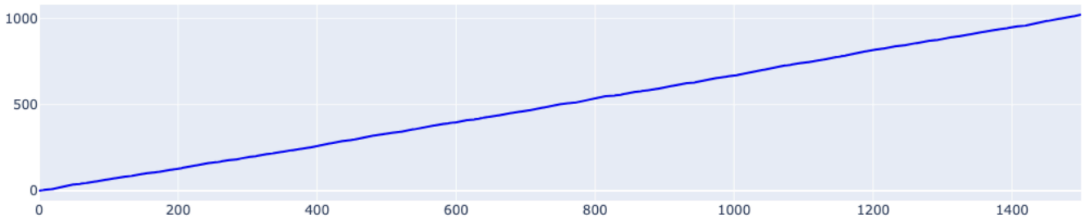
UCB

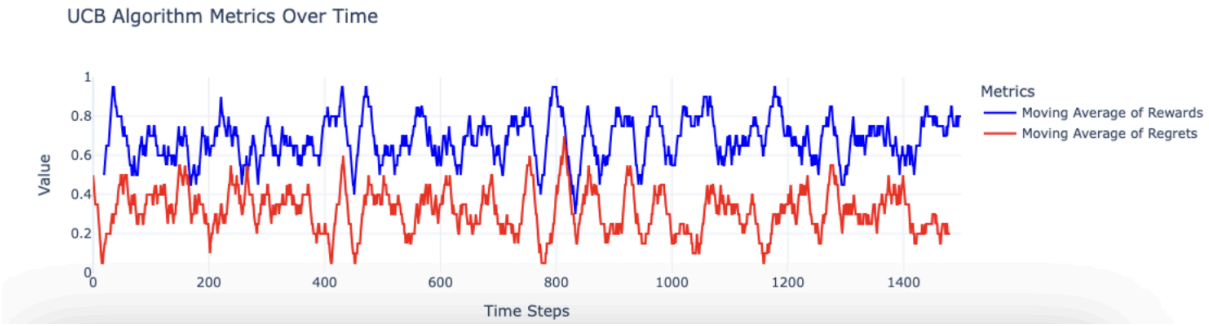
We can see a similar behavior to Thompson Sampling here. The algorithm is able to identify both optimal actions and tries both of them.
We calculate the UCB values like this:

```
ucb_values = np.zeros(self.agent.n_arms)
for arm in range(self.agent.n_arms):
   avg_reward = q_values[arm]
   confidence_bound = math.sqrt((2 * math.log(total_action_count)) /
action_counts[arm])
   ucb_values[arm] = avg_reward + confidence_bound

action = np.argmax(ucb_values)
```

```
Mean Reward: 0.682, 95% CI: [0.658405877553211, 0.7055941224467891]
Mean Regret: 0.318, 95% CI: [0.29440587755321096, 0.34159412244678905]
```







All three of the algorithms perform very well. Thompson Sampling and UCB give more chances to other actions while e-greedy converges to one single optimal action.

Q2

There are two types of AD, type1 and type2.
Here are the possible actions and their win probabilities:

```
self.n_arms = 4
self.p_action_rewards = [0.3, 0.5, 0.75, 0.66]
# actions = [(type1, type1), (type2, type2), (type2, type1), (type1, type2)]
self.action_rewards = [3, 2, 3, 2, 3]
```

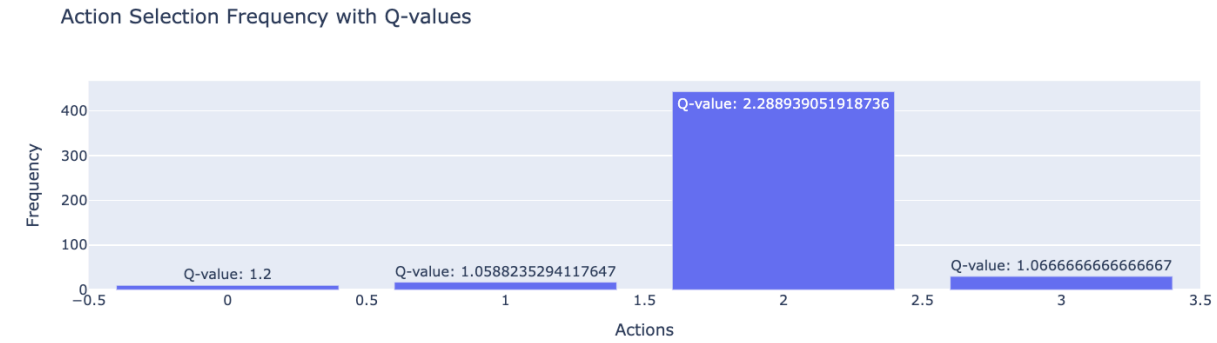
I used the Thompson Sampling algorithm for solving this problem.
Exploration and exploitation in Thompson Sampling happens as we gather more data about each action. In Thompson Sampling, we take samples from the reward distributions and act greedy upon the taken samples. This means, if the distribution of an action has a high variance (needs to be explored more), there's a higher chance for that action to be selected.
As we gather more data about the actions, variance decreases and the action selection becomes more deterministic.

A-B testing
In A-B testing, we try each arm 20 times, and select the arm with the highest reward average. Afterwards, we will act greedy upon that arm.

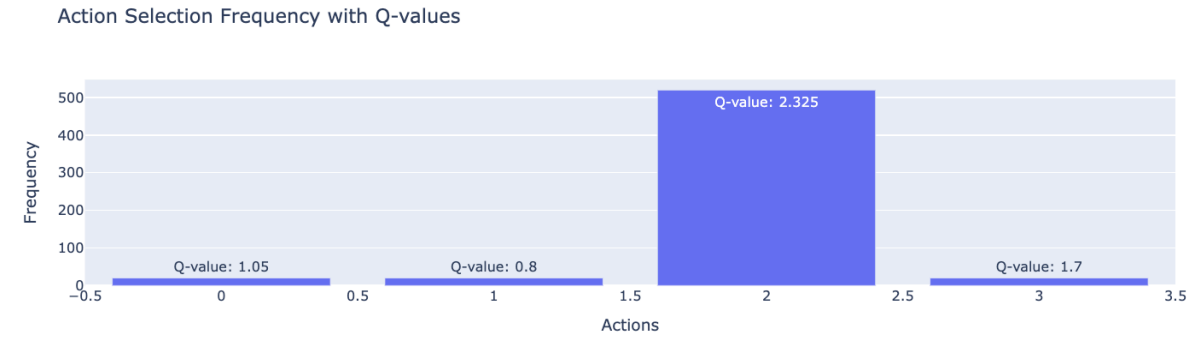
```
for arm in range(self.agent.n_arms):
    for _ in range(20):
        reward = self.agent.act(arm)
        regrets.append(3 - reward)
        rewards.append(reward)

        actions.append(arm)
        action_rewards[arm].append(reward)
        q_values[arm] = np.mean(action_rewards[arm])

best_arm = np.argmax([np.mean(rewards[i * 20:(i + 1) * 20]) for i in
range(self.agent.n_arms)])
```

Both A-B testing and Thompson Sampling find the optimal action easily. If the reward probabilities are very low for all the actions, there's a high chance for A-B testing to choose the best arm incorrectly.

```
Mean Reward: 2.152, 95% CI: [2.0364861851570226, 2.2675138148429776]
Mean Regret: 0.848, 95% CI: [0.7324861851570226, 0.9635138148429774]
```




```
Mean Reward: 2.207, 95% CI: [2.1016019645578976, 2.312191138890378]
Mean Regret: 0.793, 95% CI: [0.6878088611096218, 0.8983980354421024]
```



Q3

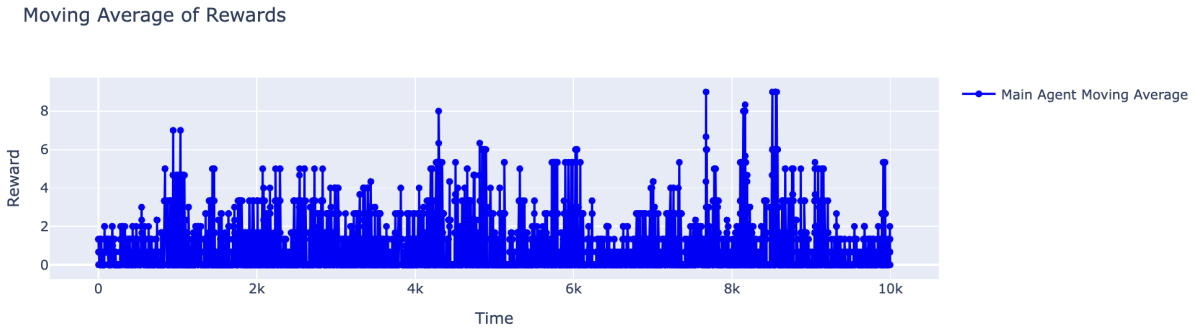
General Approach

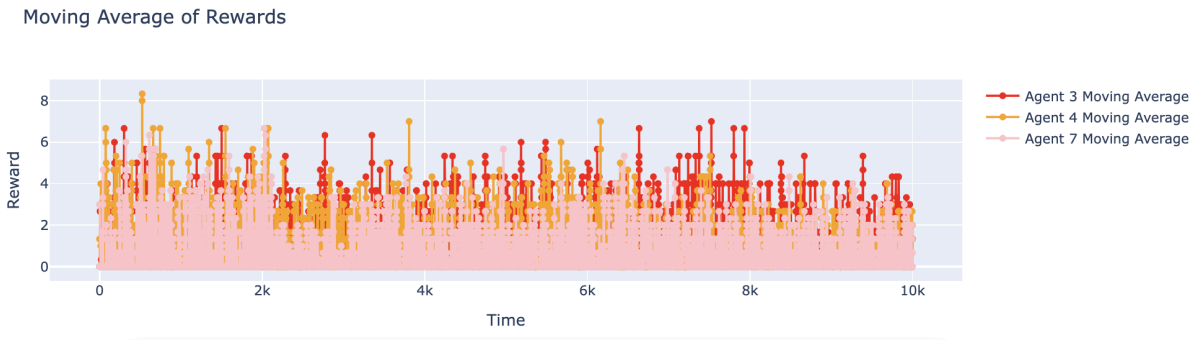
In the environment, there are several agents which follow the Thompson Sampling policy (Beta distribution) and there is one main agent which follows the E-Greedy policy.
The main agent receives the chosen actions of other agents as information but does not see their rewards. The agent acts based on it's own experience using a E-Greedy policy, but increases the probability of actions that are frequently selected by others.

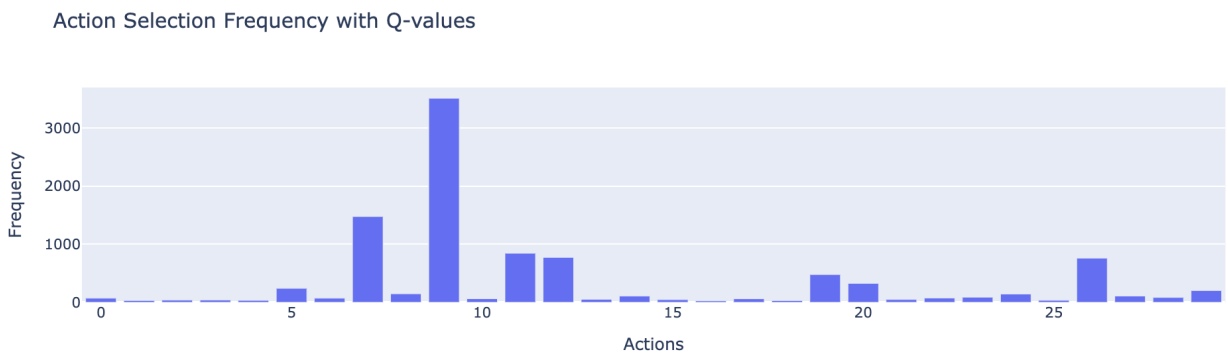
Results

I have printed the mean rewards of all the agents and we can see that our main agent stands above most of the others.

```
Main Agent: Mean = 0.8088, 95% CI = [0.7726621469462032, 0.8449378530537968]
Agent 1: Mean = 0.7936, 95% CI = [0.7584772622561398, 0.8287227377438602]
Agent 2: Mean = 0.7566, 95% CI = [0.7237911458655358, 0.7894088541344643]
Agent 3: Mean = 0.8736, 95% CI = [0.8385559942961816, 0.9086440057038185]
Agent 4: Mean = 0.7738, 95% CI = [0.7405647830542749, 0.8070352169457252]
Agent 5: Mean = 0.7109, 95% CI = [0.6798645482831274, 0.7419354517168726]
Agent 6: Mean = 0.8321, 95% CI = [0.79674547345597, 0.8674545265440299]
Agent 7: Mean = 0.5959, 95% CI = [0.568328662811547, 0.6234713371884529]
Agent 8: Mean = 0.6554, 95% CI = [0.6276228231445877, 0.6831771768554122]
Agent 9: Mean = 0.7616, 95% CI = [0.7280406986111506, 0.7951593013888495]
Agent 10: Mean = 0.8512, 95% CI = [0.816849098062145, 0.8855509019378549]
Agent 11: Mean = 0.718, 95% CI = [0.6859667902956843, 0.7500332097043156]
Agent 12: Mean = 0.903, 95% CI = [0.8630464700136152, 0.9429535299863848]
Agent 13: Mean = 0.707, 95% CI = [0.6769750806855122, 0.7370249193144878]
Agent 14: Mean = 0.7365, 95% CI = [0.7043805830192501, 0.76861941698075]
Agent 15: Mean = 0.8853, 95% CI = [0.848885131521304, 0.921714868478696]
Agent 16: Mean = 0.9218, 95% CI = [0.8861167116560769, 0.957483288343923]
Agent 17: Mean = 0.6344, 95% CI = [0.6060892016873122, 0.6627107983126878]
Agent 18: Mean = 0.8994, 95% CI = [0.8630953862379345, 0.9357046137620655]
Agent 19: Mean = 0.6889, 95% CI = [0.6595601620232059, 0.718239837976794]
Agent 20: Mean = 0.6548, 95% CI = [0.6266286716879661, 0.682971328312034]
```







Implementation

There are 3 classes for this homework: `MainAgent`, `OtherAgent` and `UserSimulation`.
`MainAgent` class is our main agent, it has a function called `receive_information` this function takes in the selected actions of other agents, and saves it to be used later.
This function is called by `UserSimulation` every iteration to gather the information about other agents.

The policy is a simple Epsilon Greedy policy. The only difference is that we add the “normalized action counts” to the Q-values too. Here is the code:

```
def act(self):
    if random.random() < self.epsilon:
        action = random.randint(0, self.n_arms - 1)
    else:
        normalized_other_counts = self.other_agents_actions /
np.sum(self.other_agents_actions + 1e-5)
        combined_values = self.Q_values + normalized_other_counts
        action = np.argmax(combined_values)

    reward = self.__act(action)
    self.action_counts[action] += 1
    self.Q_values[action] += self.learning_rate * (reward - self.Q_values[action])

    return reward, action
```

The `OtherAgent` class is almost a copy of the agent from the previous homework questions. It follows a Thompson Sampling policy and does nothing more.

The `UserSimulation` class is responsible for stimulating and running the environment. At each iteration, it first runs all the other agents and saves the selected actions.

Then, it calls the MainAgent.recieve_information() and passes the information to our main agent, and finally runs the MainAgent itself.

```
for _ in range(n_iteration):

    # run other agents first
    other_agent_actions = []
    for agent in self.other_agents:
        action, reward = agent.act()
        other_agent_actions.append(action)

        try:
            other_agent_rewards[agent.name].append(reward)
        except KeyError:
            other_agent_rewards[agent.name] = [reward]

    self.main_agent.receive_information(other_agent_actions)

    reward, action = self.main_agent.act()
    main_agent_action.append(action)
    main_agent_rewards.append(reward)
```