CS6700 : Reinforcement Learning
Programming Assignment #1
Report

Sai Vinay G
CE17B019
Indian Institute of Technology, Madras

March 3, 2019

Problem 1

Implementation of ϵ-greedy algorithm on 10-armed testbed.

- We initialize the expected values of each arm to zero and maintain the number of times each arm is pulled.
- We select the arm with highest expectation with $(1 - \epsilon + \epsilon/k)$ probability and remaining arms with ϵ/k probability.

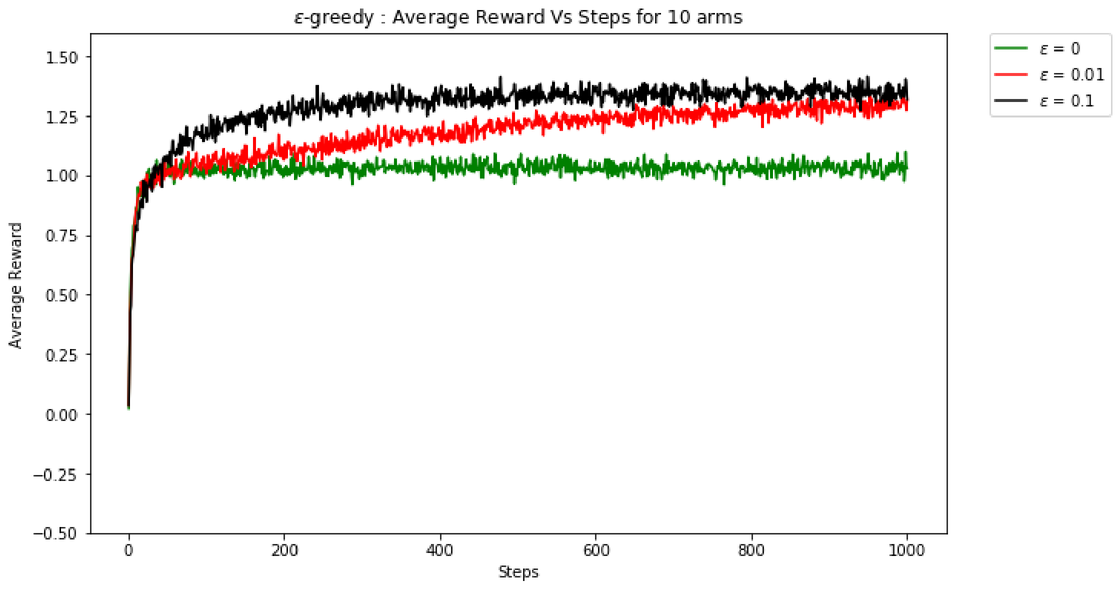

Figure 1: Average performance of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

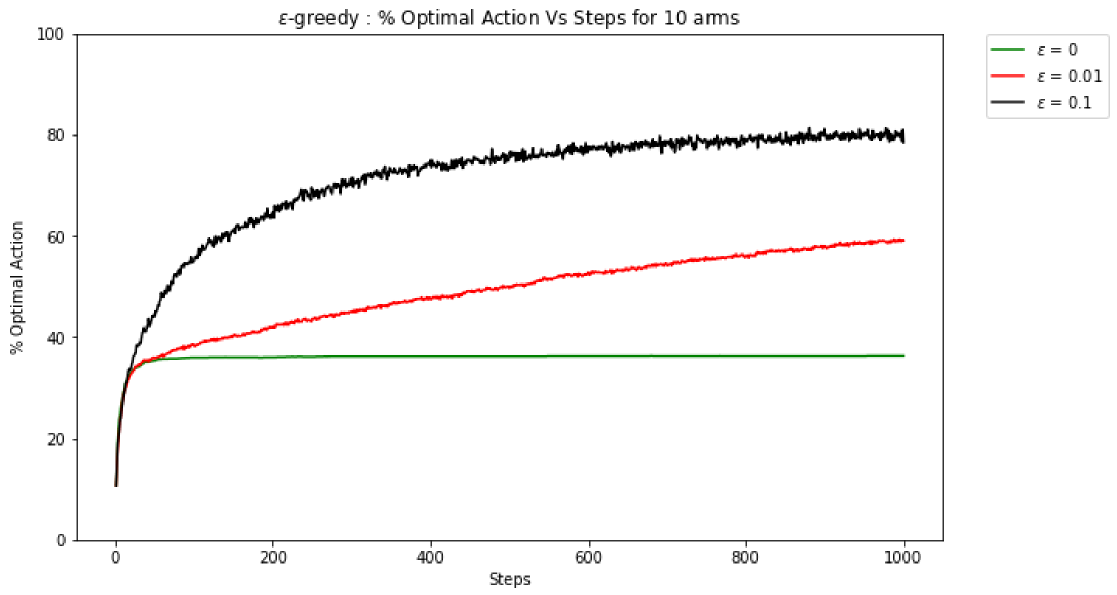

Figure 2: % Optimal arm selection of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

In the above plots we try reproduce the results obtained in the textbook.

- In Figure 1 we compare average rewards obtained by three values of epsilons (i.e, 0, 0.01, 0.1). We can see that highest average reward is for $\epsilon = 0.1$, then $\epsilon = 0.01$ and at last $\epsilon = 0$.
  - In the case of $\epsilon = 0.1$, it quickly explores all the arms and finds the optimal arm so, its average reward is high than the case of $\epsilon = 0.01$, it slowly explore all the arms so, the average reward is lesser than $\epsilon = 0.1$ case and the case of $\epsilon = 0$ (greedy method), the average reward is to low because it doesn't explore at all ,it only exploits the arm with highest expected reward. So, it may get stuck at an sub optimal arm.
  - We can see that the agents with $\epsilon = 0.1$ and $\epsilon = 0.01$ converge to the arm with highest expectation, whereas the agent with $\epsilon = 0$ converges to a sub optimal arm.
  - As we are doing it for only 1000 steps $\epsilon = 0.01$ has lesser average reward than the case with $\epsilon = 0.1$. If we do it for more steps(say 10,000 steps with 10 arms), it($\epsilon = 0.01$) will get higher reward than the case with $\epsilon = 0.1$ as, even after finding optimal arm $\epsilon = 0.1$ will still explore with higher probability. We can also see from the plot, the curve for $\epsilon = 0.01$ has higher slope than for $\epsilon = 0.1$
- In Figure 2 we compare % of times optimal is selected at each step for the same above values of ϵ's.
  - For $\epsilon = 0.1$ we can see that it takes optimal action 80% of the time at the end, whereas $\epsilon = 0.01$ takes optimal action 60% of the time at the end and $\epsilon = 0$ takes optimal action for only around 35% of the time.
  - The agent with $\epsilon = 0.1$ is quickly exploring so in 2000 bandit problems many of them quickly find the optimal arm so it has around 80%. In the case of $\epsilon = 0.01$ the agent explore slowly so in all bandit problems they wont be able to find the optimal arm quickly. In case of $\epsilon = 0$ greedy agent only some 35% of the bandits problems get the optimal arm as they select randomly initially. (It is 35% because if initially they pick an arm with negative reward, they choose for another arm in which they may pick optimal arm (Hence its around 35%, not around 10%)

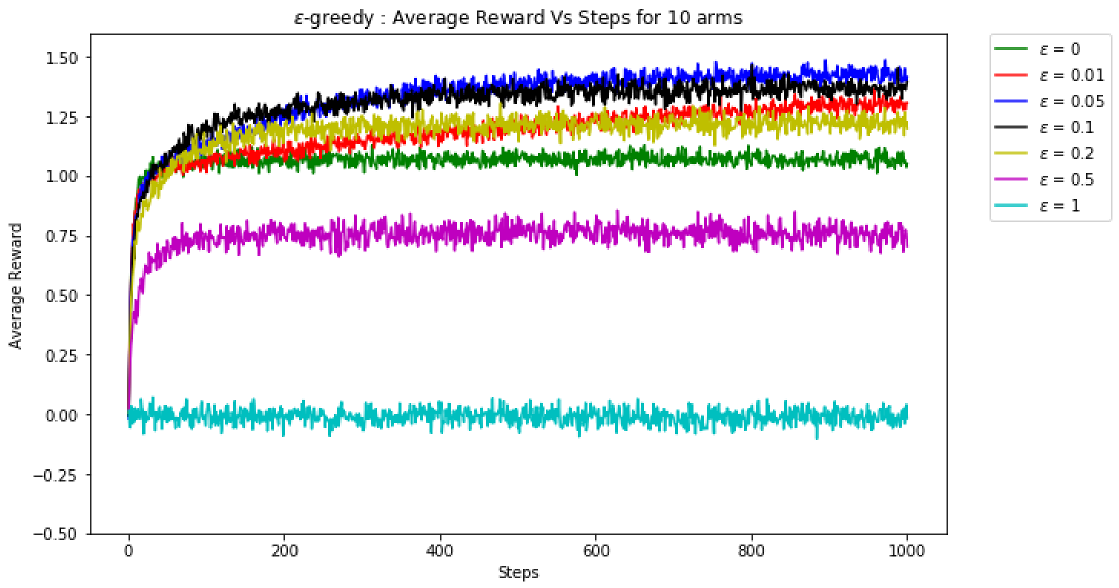

Figure 3: Average performance of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

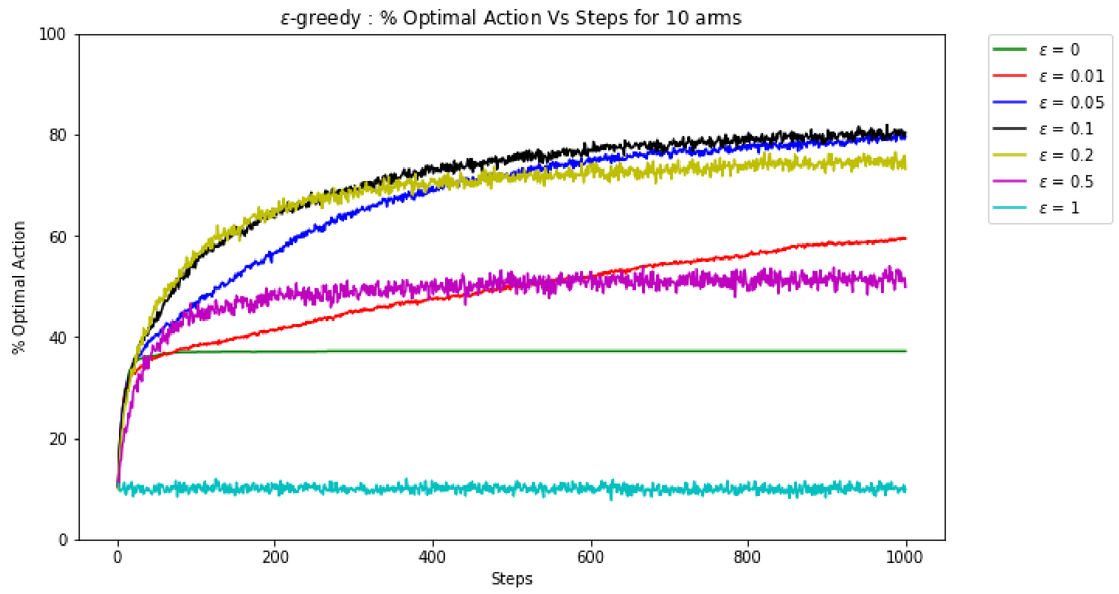

Figure 4: % Optimal arm selection of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

The above plots shows the comparison between various values of ϵ's = 0, 0.01, 0.05, 0.1, 0.2, 0.5, 1.

Here we get maximum reward for $\epsilon = 0.05$ and the maximum %optimal arm selection for $\epsilon = 0.1$

Problem 2

In this problem we Implement softmax action selection method using the Gibbs distribution on the 10-armed testbed.

- We get the preference of action by dividing Q/T where, Q = expected reward for pulling and arm and T = temperature.
- We get the probabilities of selecting an action using softmax(preferences).

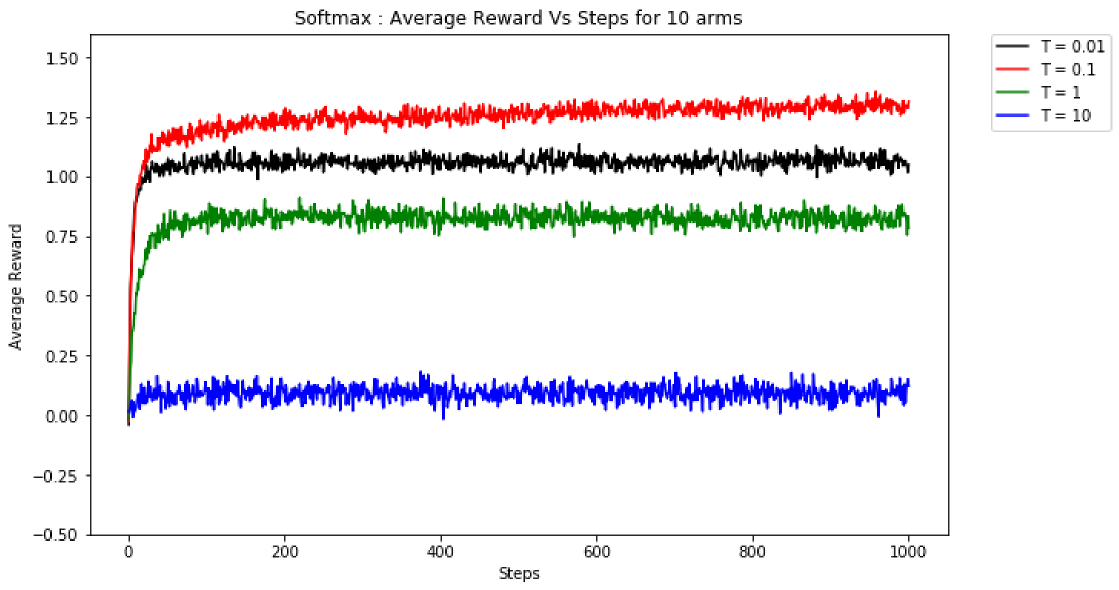

Figure 5: Average performance of Softmax method on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

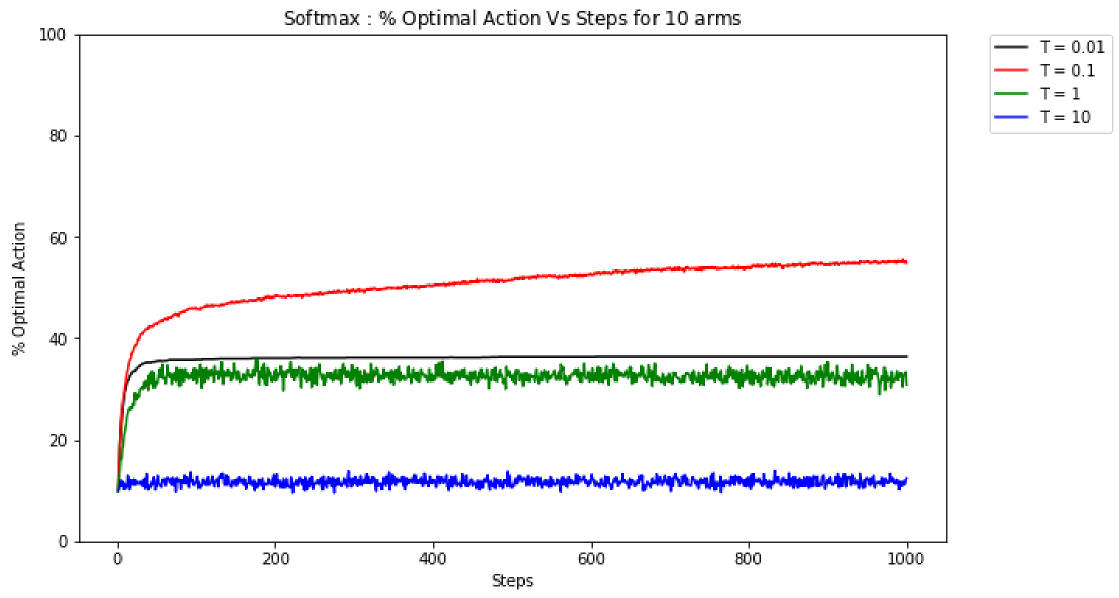

Figure 6: % Optimal arm selection of Softmax method on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

- If the temperatures are very high all the arm have the same probability, so it always pick random actions.

If the temperatures are close to 0 then it acts like greedy arm (as the greedy has more on numerator than on denominator). The values in between have a trade off between exploration and exploitation.

- From Figure 5 we can see that T = 0.1 has highest average reward, next with T = 0.01, then with T = 1 and then T = 10.
  - From above we can say that T = 10 acts as random arm picker so, it has average value as 0.
  - T = 0.1 seems to be the ideal case where exploration and exploitation are well balanced so it has higher reward when compared to other T values.
  - Similarly we get the plots for % optimal action

Problem 3

In this problem we implement the UCB1 algorithm.

- We first pull each arm once to get the initial estimates of each arm. Then using UCB1 theorem we find the optimal arm.

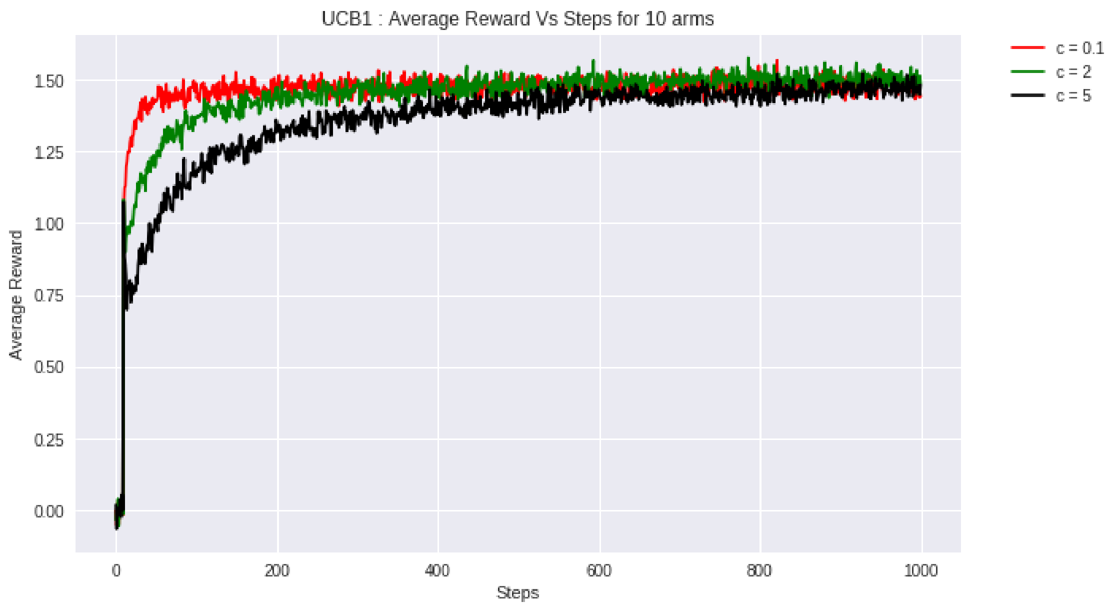

Figure 7: Average performance of UCB1 method on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

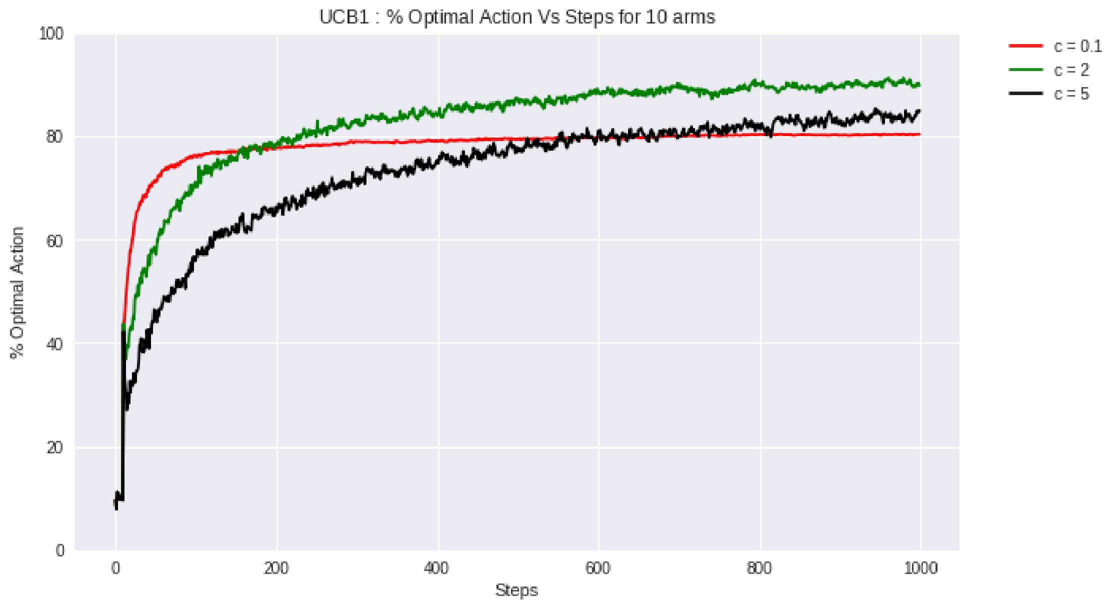

Figure 8: % Optimal arm selection of UCB1 method on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

- We can see that using c = 2 we get slightly more reward than with c = 5, than c = 0.1 .
- This is because for very low values of c, it behaves like a greedy algorithm, for high values of c, it there is a possibility of selecting sub optimal arms as their bounds maybe high.

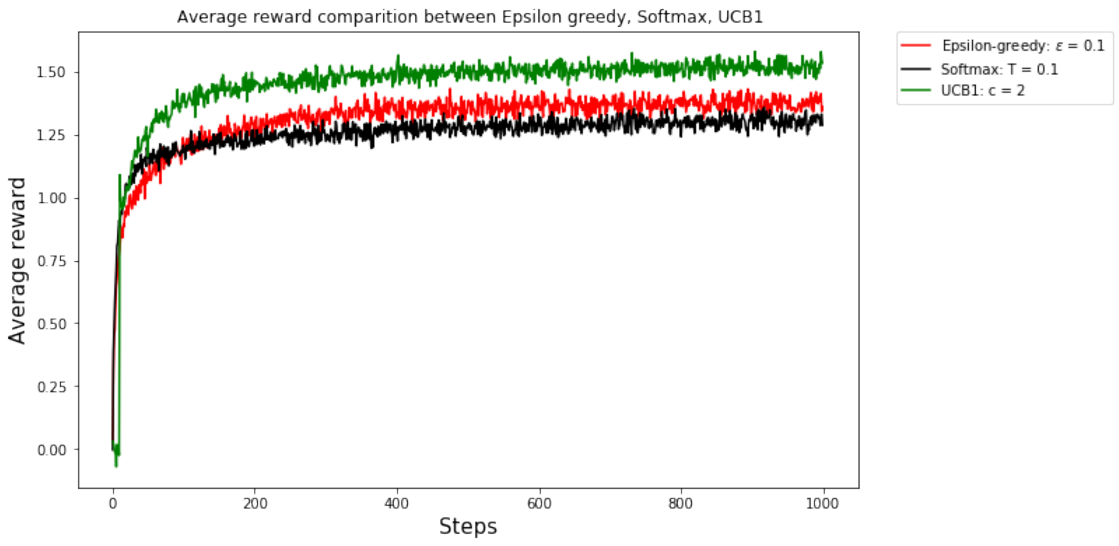

Figure 9: Comparison of Average performance between ϵ-greedy, Softmax, UCB1 methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

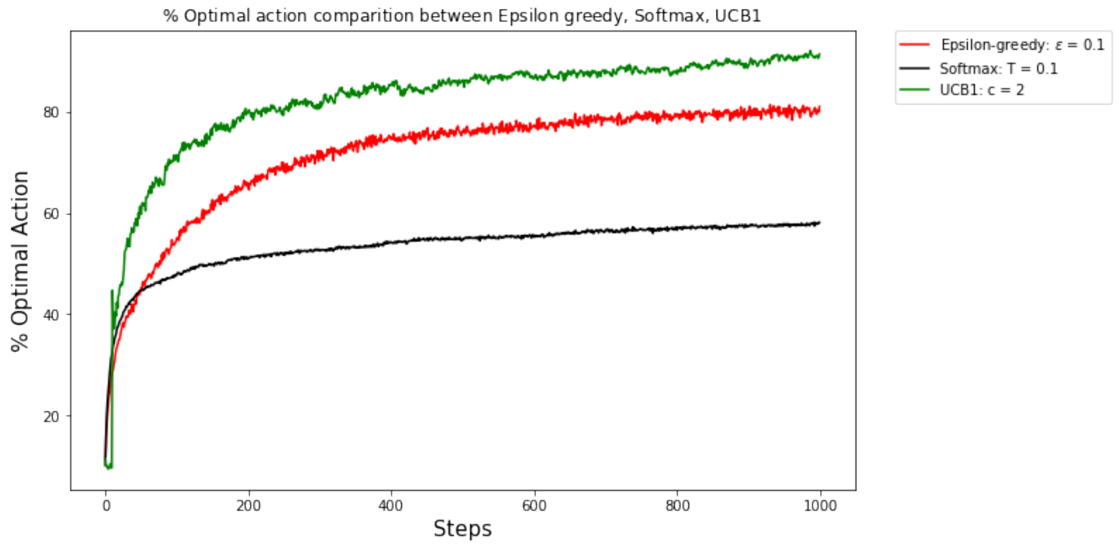

Figure 10: Comparison of % Optimal arm selection between ϵ-greedy, Softmax, UCB1 methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems

- We can see from comparison that UCB1 performs better than ϵ-greedy which is better than softmax.
- This is because as in case of ϵ-greedy we explore random action we though some have very low expectation, which we can over come using UCB, as selects non-greedy actions based on their potential to be actually optimal and softmax is kind of greedy scenario as the arms having high expectation are given more probability of picking, so it is the lowest of all.

Problem 4

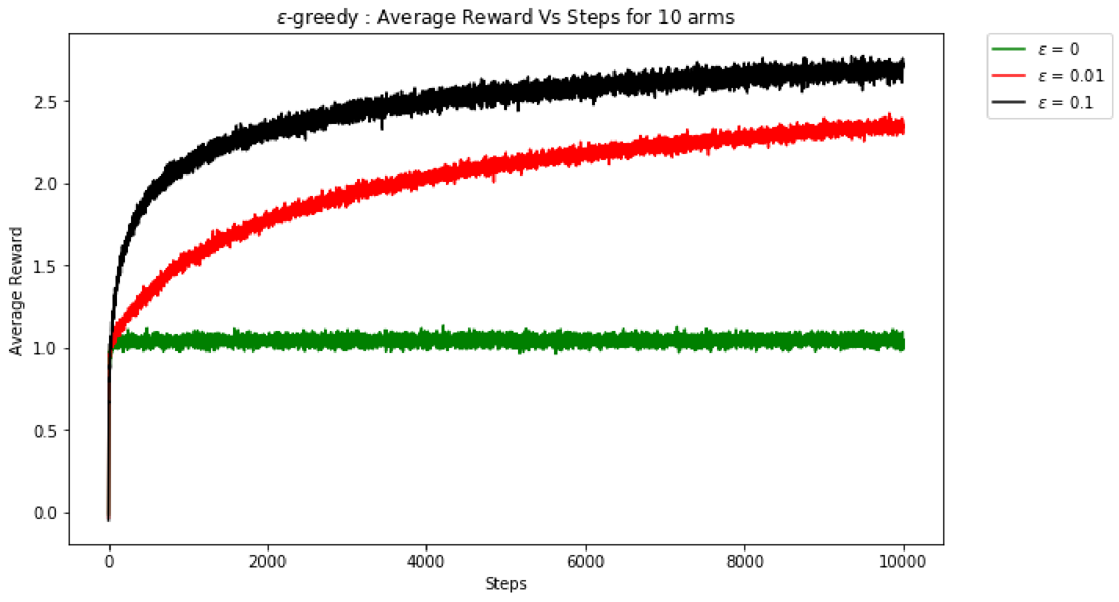

Figure 11: Average performance of ϵ-greedy action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

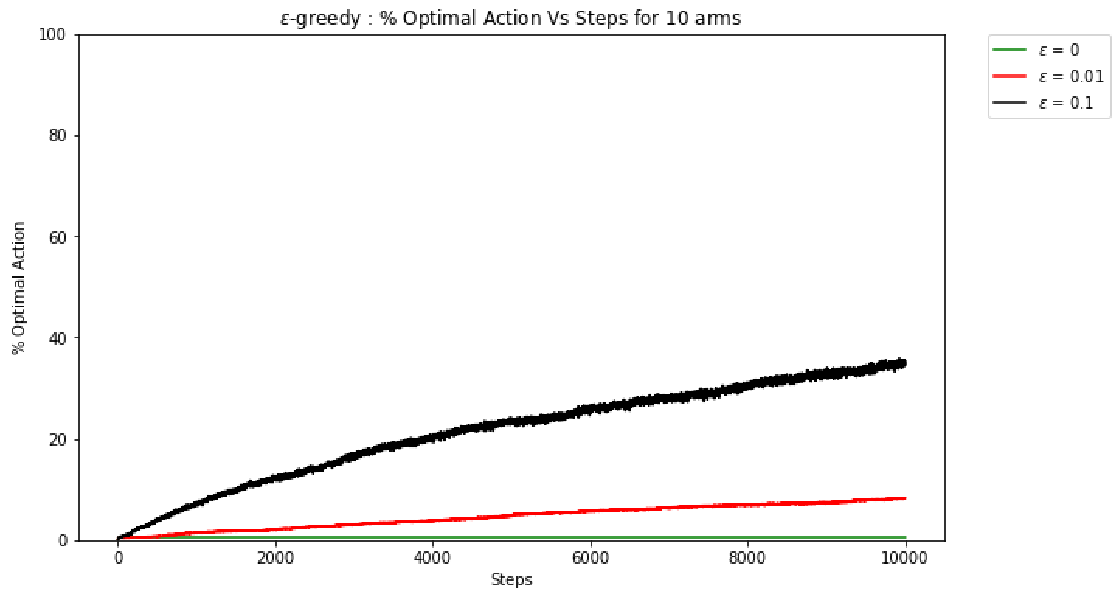

Figure 12: % Optimal arm selection of ϵ-greedy action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

- We can see that the reward for $\epsilon = 0$ doesn't change, whereas, the average reward for both $\epsilon = 0.01$ and $\epsilon = 0.1$ has significantly increased.
- This is because we have 1000 arms its very difficult to explore so the arm with $\epsilon = 0.1 outperforms \epsilon = 0.01$ as the latter explores slowly.But the $\epsilon = 0$ case preforms greedily which almost doesn't depend on number of arms.

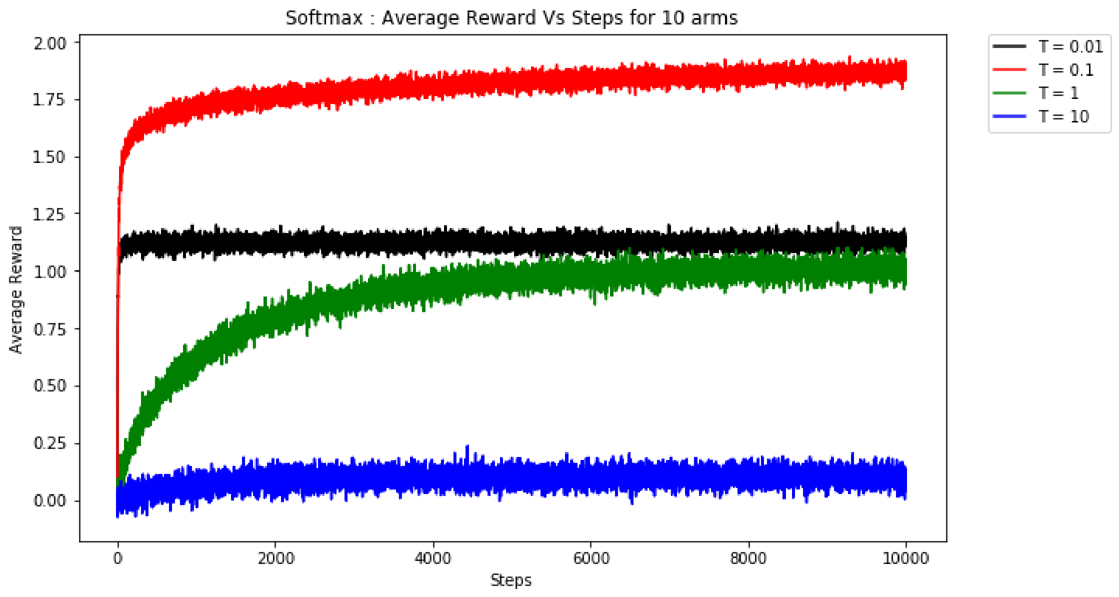

Figure 13: Average performance of Softmax action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

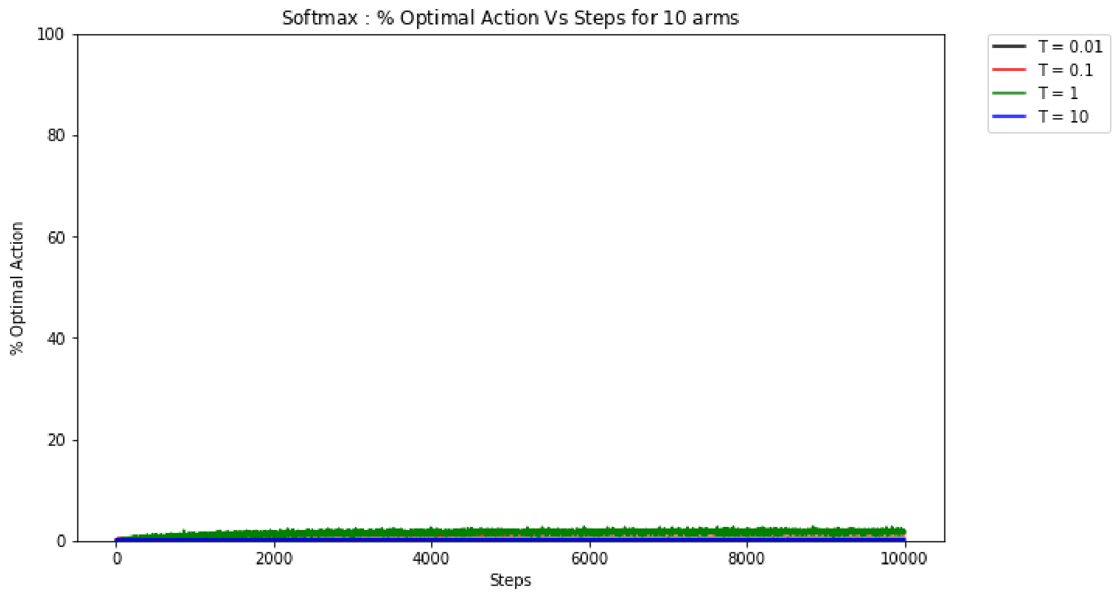

Figure 14: % Optimal arm selection of Softmax action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

- We can see that the average reward for T = 0.01, 0.1, 10 hasn't changed much but there is lot of variation for T = 1.
- This may be because in softmax we have probabilities, as there are 1000 arms the probabilities get distributed so there is less probability for even to optimal to get selected frequently.

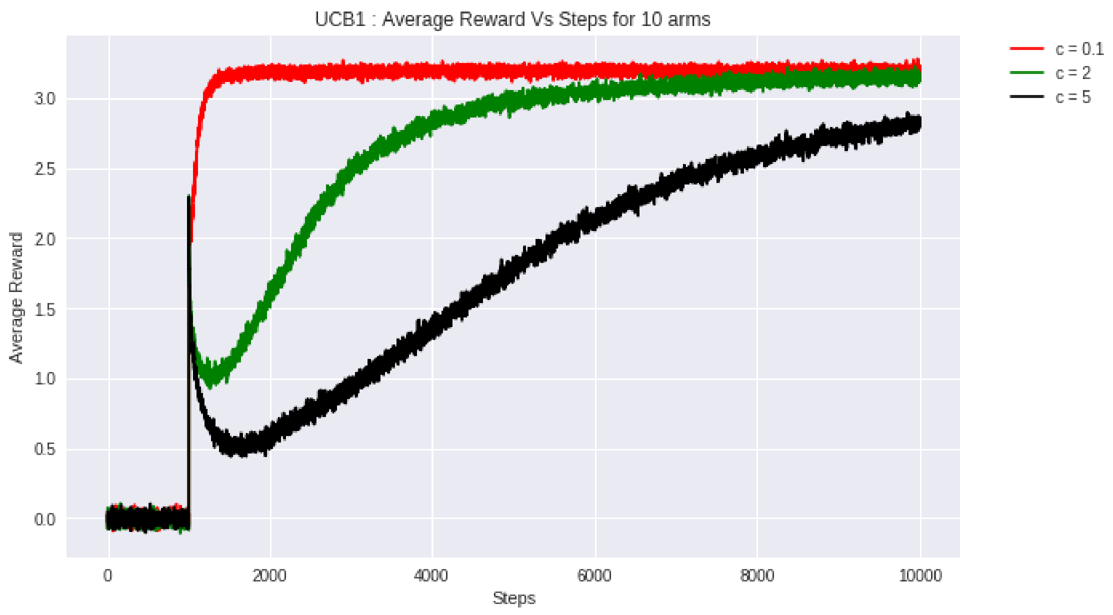

Figure 15: Average performance of UCB1 action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

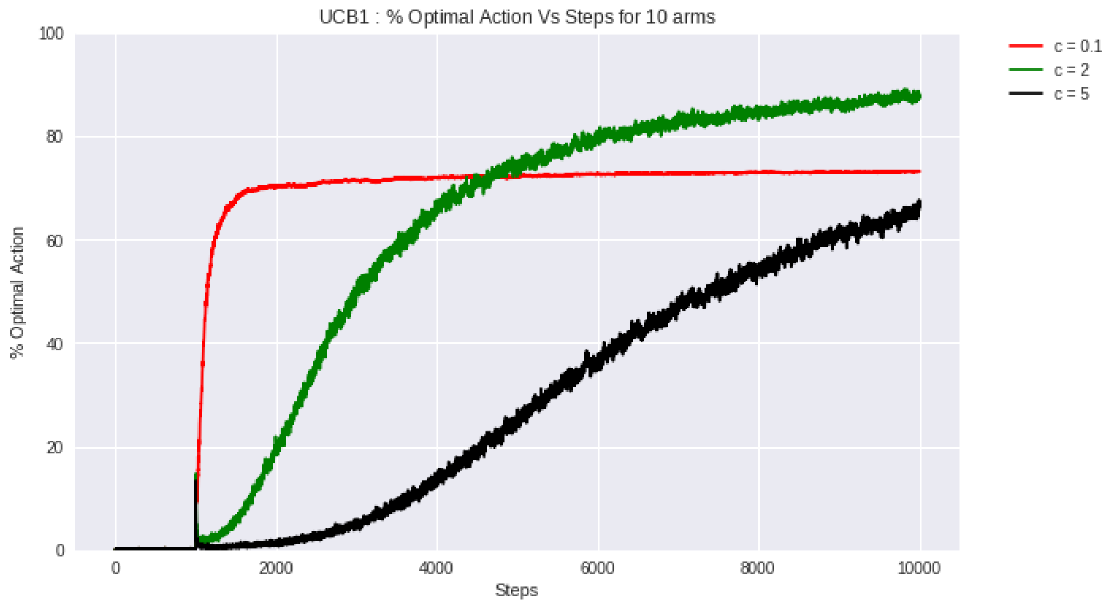

Figure 16: % Optimal arm selection of UCB1 action-value methods on 1000 arms and 10000 steps. These data are averages over 2000 runs with different bandit problems

- Here we see that the average reward is higher for c = 0.1 which was least in case of 10 arms, and the average reward for c = 5 has much less as compared to the other c values.
- The rewards for each value has significantly increased when compared to the earlier case of 10 armed testbed.
- As we have 1000 arms there is high possibility that many arms have high expected values so as UCB picks each arm initially once, it can get to know approximate expected of all the arm, so it was able to get higher rewards as compared to softmax rewards which hasn't changed much.

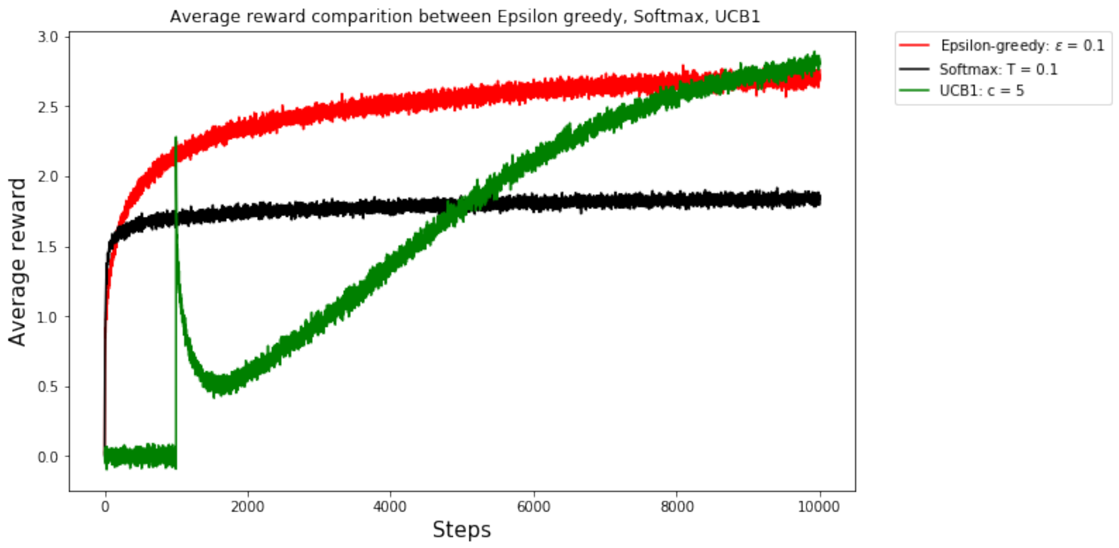

Figure 17: Comparison of Average performance between ϵ-greedy, Softmax, UCB1 methods on 1000 arm and 10000 steps. These data are averages over 2000 runs with different bandit problems

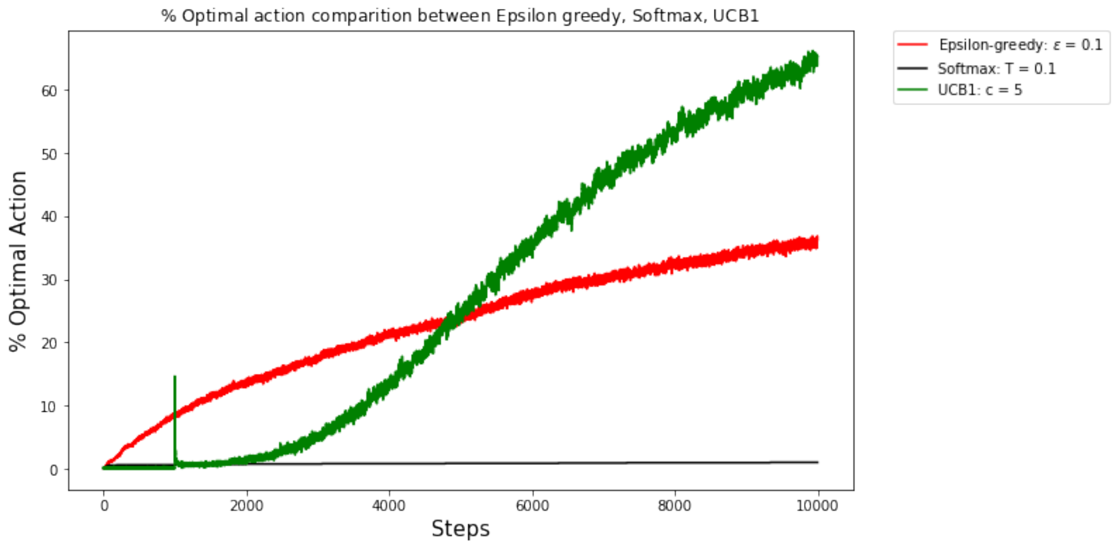

Figure 18: Comparison of % Optimal arm selection between ϵ-greedy, Softmax, UCB1 methods on 1000 arm and 10000 steps. These data are averages over 2000 runs with different bandit problems

- Here ϵ-greedy preforms much better than UCB in the start and at the end UCB crosses ϵ-greedy. In the case of softmax the probability of selecting the optimal arms reduces significantly so it doesn't get much higher reward.

Problem 5

A contextual 3-arm bandit problem, used to suggest appropriate Ad to the users who visits the company website.

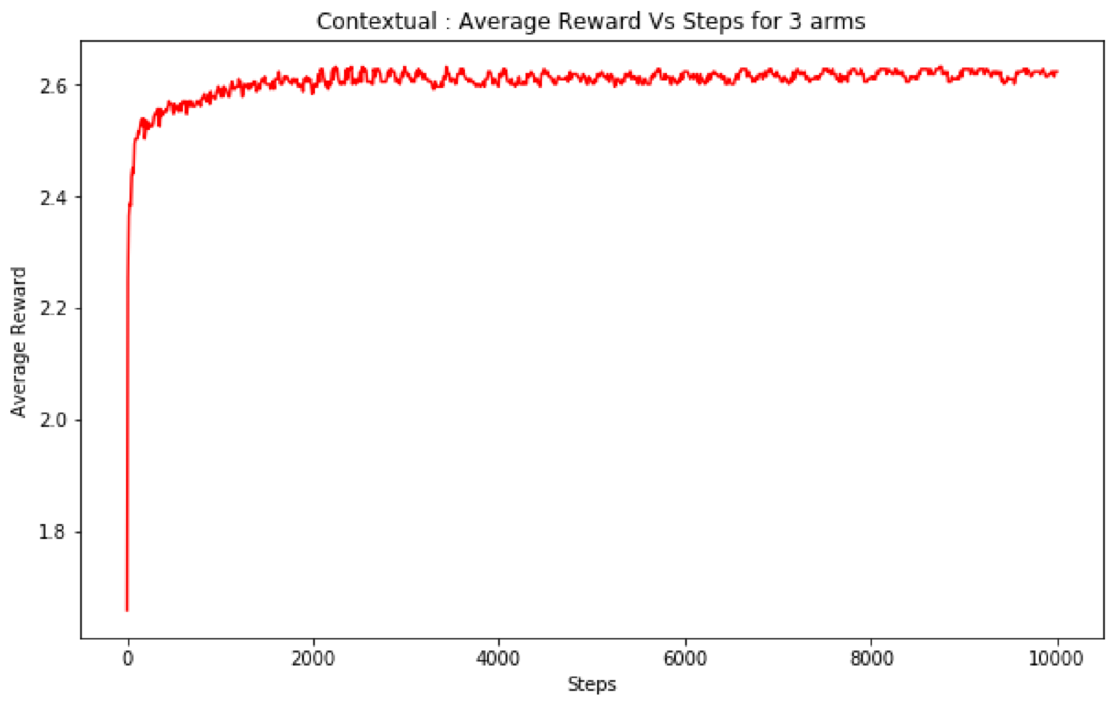

Figure 19: Plot of training curve : Contextual bandit with 3 arms.

- In this problem we use a learnable weight matrix W (4x3), which is multiplied with the vector representation of the user(4x1) to get the preference values of the user(3x1), using which we get softmax policy to select an appropriate Ad.
- We update the parameters of the weight matrix W using the REINFORCE algorithm, as given below.

```
for i in range(W.shape[0]):
    for j in range(W.shape[1]):
        if j == action:
            W[i,j] = W[i,j] + alpha*(reward-baseline)*(1-policy[action])*stateVec[i]
        else:
            W[i,j] = W[i,j] - alpha*(reward-baseline)*policy[j]*stateVec[i]
```

- The average reward in the plot is obtained between every log_interval (In this problem its 10) number of steps, by evaluating on test data, on 450 samples.
- The average reward at the end of training process was around 2.622 . Where the actual rewards assigned for picking a right advertisement is +3 and for picking any other advertisement is +1.
- We can see that the reward quickly converges to around 2.6, because of the REINFORCE algorithm. The rewards don't increase much after that because, we have used only 12 parameters to get the representation of all the users.

References

[1] matplotlib. https://matplotlib.org/tutorials/introductory/pyplot.html.

[2] Numpy. http://www.numpy.org/.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning.* MIT press, 2016.