CS6700 : Reinforcement Learning

Programming Assignment 1

Akshit Kumar
EE14B127

24th February 2017

Contents

1 Introduction

1.1 Goal

The goal of this assignment is to evaluate differenet bandit algorithms on an N-armed bandit testbed. This report analyses and presents the results obtained by simulating an N-armed bandit problem on 3 types of Action-Value Methods:

- ϵ-greedy
- Sampling from softmax distribution
- UCB-1 Algorithm

Each of the above algorithms are first run on a test-bed of 10-armed bandits with the results averaged over 2000 different bandit problems for 1000 time steps. In addition to this, we also compare the three algoritms on a test-bed of 1000 arms and see how the algorithm performs for the three algorithms perform by testing it on different time-steps.

1.2 Implementation Details

1.2.1 True Expected Payoff of Arms

For the purposes of experimental analysis, the true expected payoff of the arms for different bandit problems is sampled from a standard normal distribution offset by a true reward value. Therefore the true expected payoff for each arm is

$$q_*(a) = TrueReward + \mathcal{N}(0,1)$$

1.2.2 Initialization of Estimates of Expectation of Arms

Initially all the estimates of the expectation of arms is set to zero. Therefore we have

$$Q_0(a) = 0 \qquad \forall a \in \mathcal{A}$$

1.2.3 Sampling of Rewards for Arms

The rewards for each of the arms at each time step in each of the bandit problems is sampled from a standard normal distribution offset by the true expected payoff of that arm. Therefore we have

$$R_i = q_*(a) + \mathcal{N}(0,1)$$

where R_i is the reward at time step i and $\mathcal{N}(0,1)$ adds noise to the true expected payoff for that action.

1.2.4 Update of the Estimate of Arms

For updating the estimates of the arm, incremental updates are made and the update equation is given as

$$Q_{t+1} = Q_t + \frac{1}{t}[R_t - Q_t]$$

2 Action-Value Algorithms

In this section we discuss the algorithms of the 3 types of action-value methods mentioned above.

2.1 ϵ-Greedy

Algorithm 1 ϵ-greedy Algorithm

```
1: procedure ϵ-GREEDY PROCEDURE
2:     for each action $a \in \mathcal{A}$ do
3:         Q(a)←0
4:         N(a) ← 0
5:     end for
6:     loop
7:         if sample from $\mathcal{N}(0,1) > \epsilon$ then
8:             $\mathcal{A} \leftarrow \text{argmax}_a Q(a)$
9:         else
10:            $\mathcal{A} \leftarrow$ randint(0,# of arms)
11:        end if
12:        $\mathcal{R} \leftarrow getReward(\mathcal{A})$
13:        $N(\mathcal{A}) \leftarrow N(\mathcal{A}) + 1$
14:        $Q(\mathcal{A}) \leftarrow Q(\mathcal{A}) + \frac{1}{N(A)}[R - Q(A)]$
15:    end loop
16: end procedure
```

This algorithm introduces some randomness in selection of action to take at step $t+1$ which allows us to explore more actions and not settle for a suboptimal arm. At each step, with probability ϵ , a random exploratory move is played and with probability $1-\epsilon$, a greedy action selection is made based on the current estimates of the expectation of the arms.

2.2 Sampling from softmax distribution

Algorithm 2 Sampling from Softmax Distribution

```
    procedure SAMPLING FROM SOFTMAX DISTRIBUTION
2:      for each action $a \in \mathcal{A}$ do
            Q(a) ← 0
4:          N(a) ← 0
        end for
6:      loop
            $\mathcal{A} \sim \{\frac{e^{Q_t(a_i)/\tau}}{\sum_a e^{Q_t(a)/\tau}}\}$
8:          $\mathcal{R} \leftarrow getReward(\mathcal{A})$
            $N(\mathcal{A}) \leftarrow N(\mathcal{A}) + 1$
10:         $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
        end loop
12: end procedure
```

In this algorithm, we sample from a softmax distribution which is given by

$$P_t = \frac{e^{Q_t(a_i)/\tau}}{\sum_a e^{Q_t(a)/\tau}}$$

where τ is the temperature parameter.

2.3 Upper Confidence Bound

Algorithm 3 Upper Confidence Bound

```
 1: procedure UCB-1
 2:     for each action $a \in \mathcal{A}$ do
 3:         play action $a$
 4:         $\mathcal{R} \leftarrow getReward(\mathcal{A})$
 5:         $Q(a) \leftarrow \mathcal{R}$
 6:     end for
 7:     loop
 8:         $\mathcal{A} \leftarrow \text{argmax}_a Q(a) + \sqrt{\frac{2lnN}{T_i(n)}}$
 9:         $\mathcal{R} \leftarrow getReward(\mathcal{A})$
10:         $N(\mathcal{A}) \leftarrow N(\mathcal{A}) + 1$
11:         $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
12:     end loop
13: end procedure
```

In this algorithm, we initially play all the actions once and then we take the subsequent actions in the remaining timesteps by being greedy with respect to the upper confidence bound on the current estimates of the actions. Therefore at each time step we choose,

$$a = \underset{a}{\text{argmax}}\, Q(a) + \sqrt{\frac{2lnN}{T_i(N)}}$$

3 Experimental Results

3.1 Plots for ϵ-greedy Action Selection for 10 arms

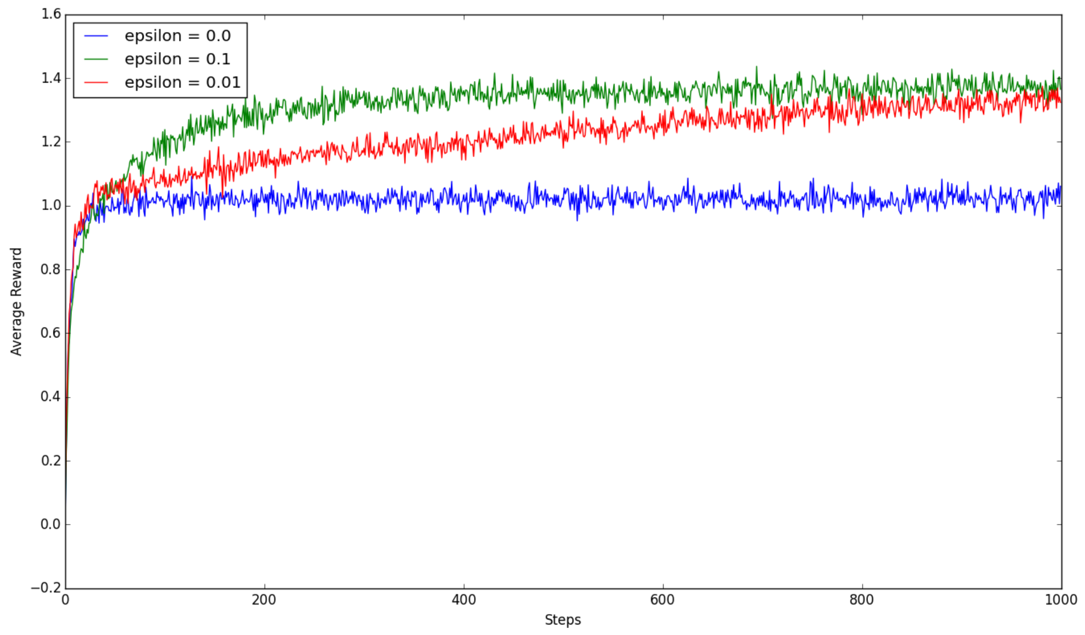

Figure 1: Plot of average reward of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems.

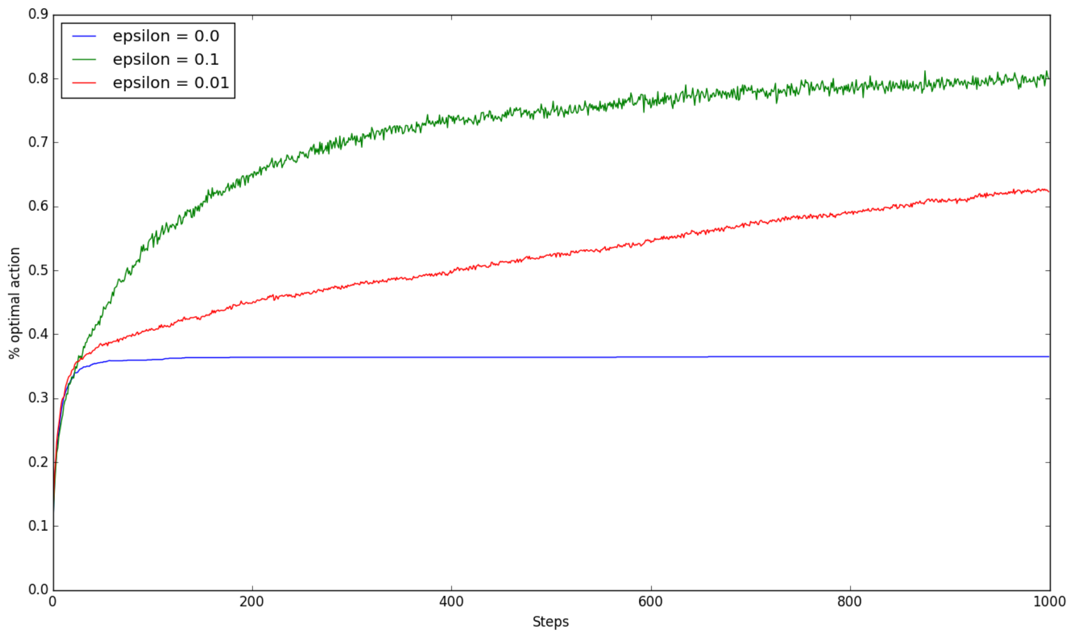

Figure 2: Plot of % of optimal action selection of ϵ-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems.

3.2 Plots for Softmax Action Selection for 10 arms

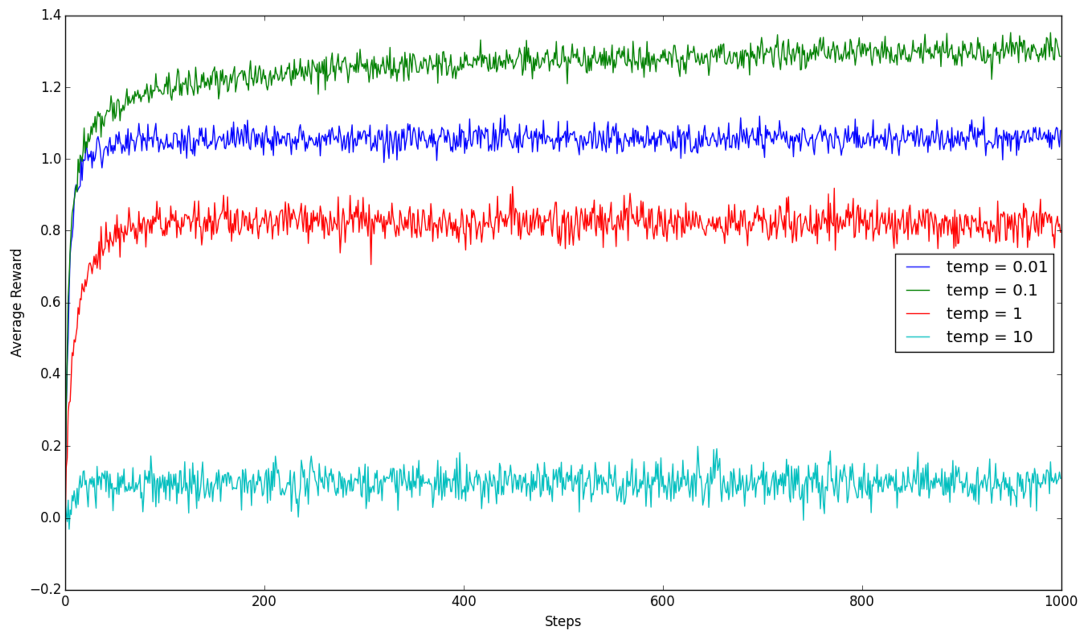

Figure 3: Plot of average reward of softmax action selection methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems.

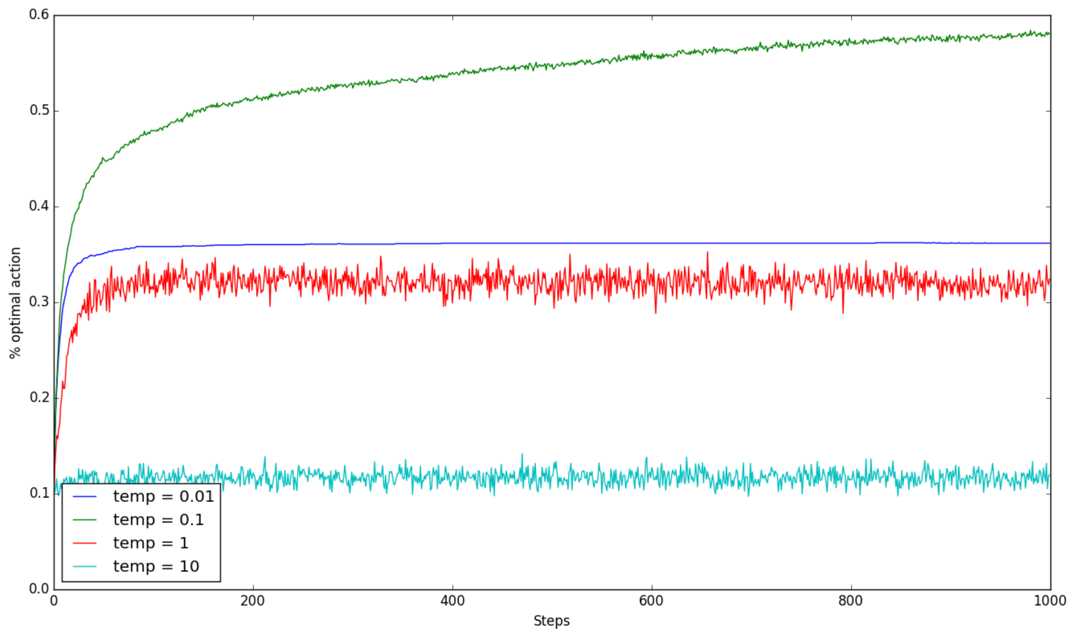

Figure 4: Plot of average reward of softmax action selection methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems.

3.3 Plots for UCB vs ϵ-greedy vs Softmax Action Selection for 10 arms

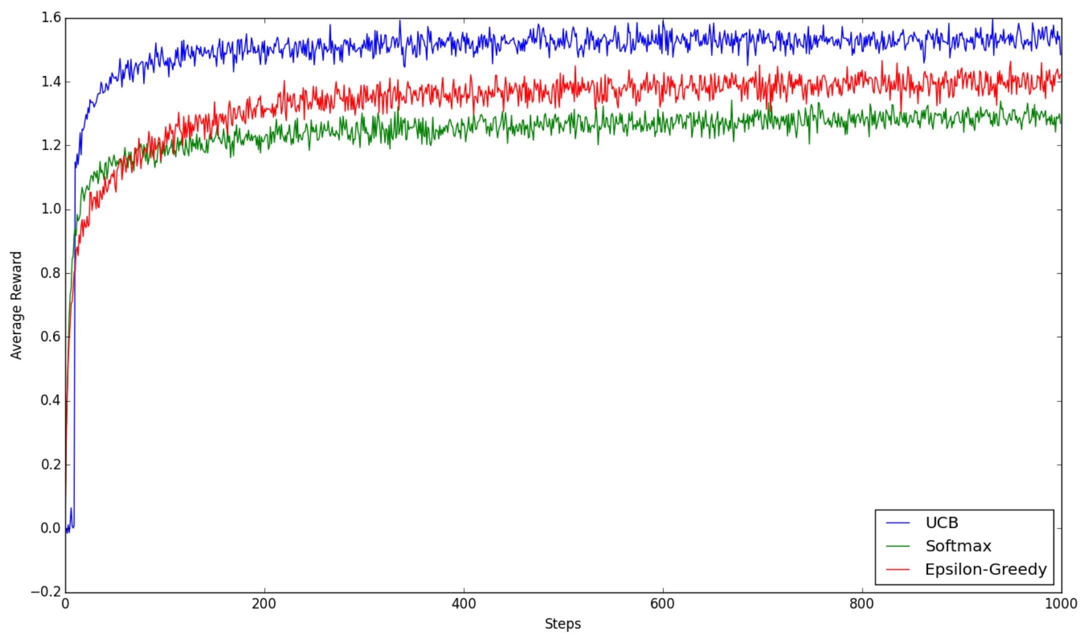

Figure 5: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 10-armed testbed over 1000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

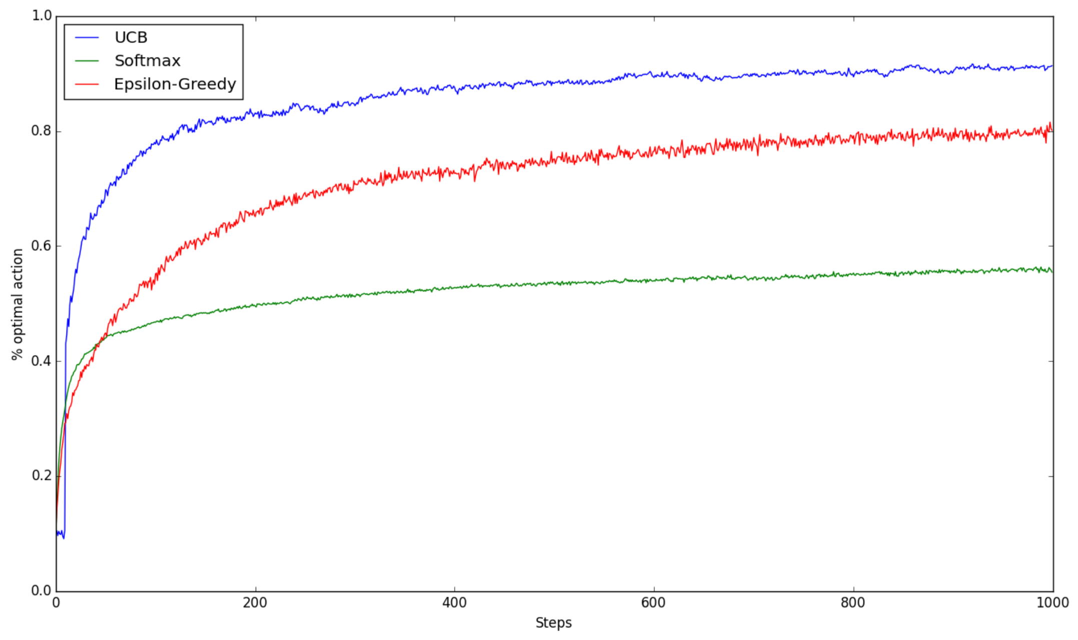

Figure 6: Plot comparing the % optimal action selection using UCB, ϵ-greedy and Softmax Action Selection on the 10-armed testbed over 1000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

3.4 Plots for UCB vs ϵ-greedy vs Softmax Action Selection for 1000-arms

3.4.1 Plots for 1000 time-steps

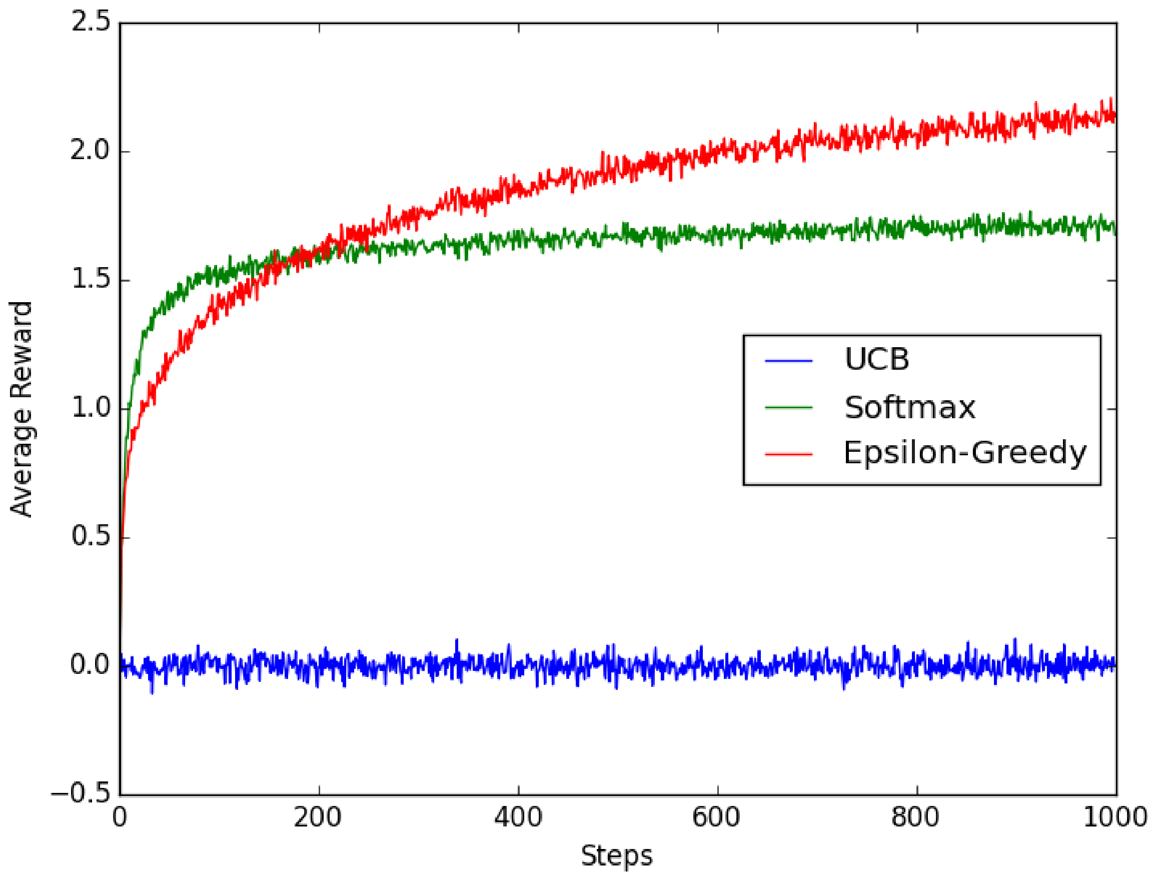

Figure 7: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 1000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

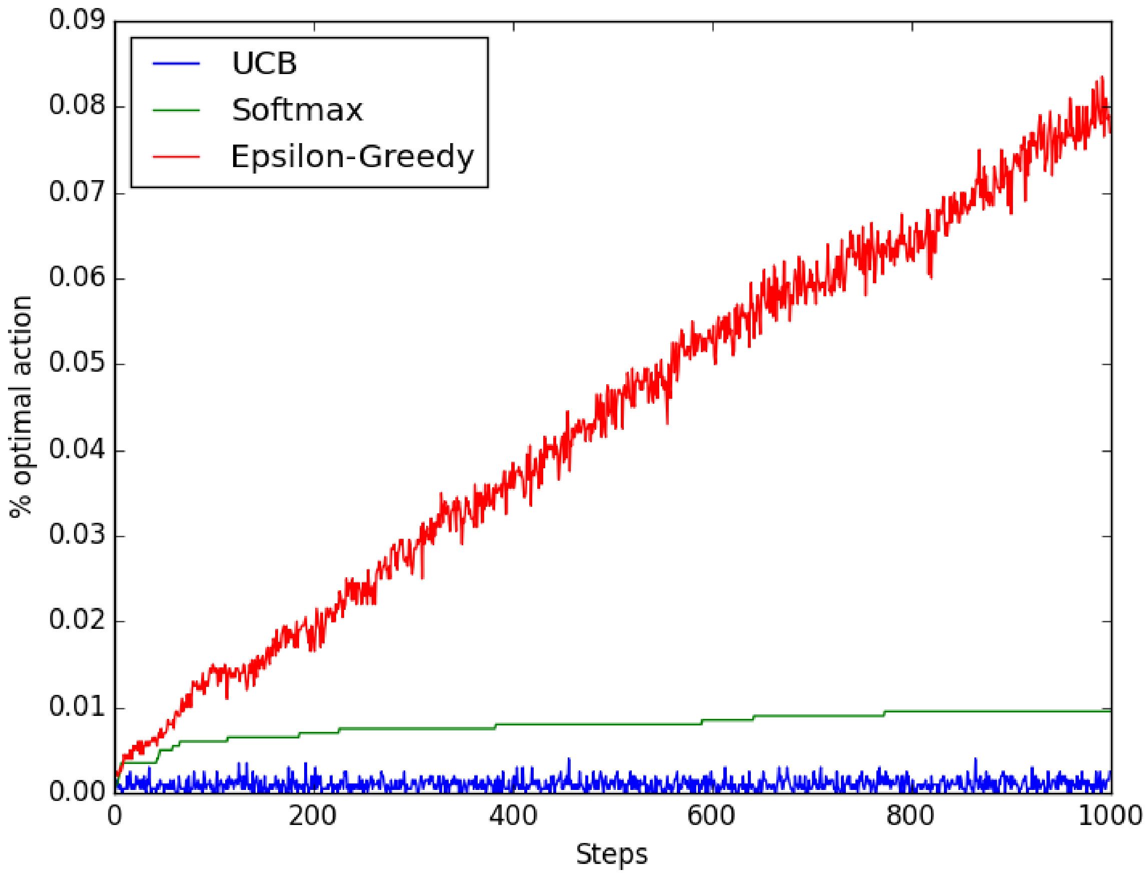

Figure 8: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 1000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

3.4.2 Plots for 10000 time-steps

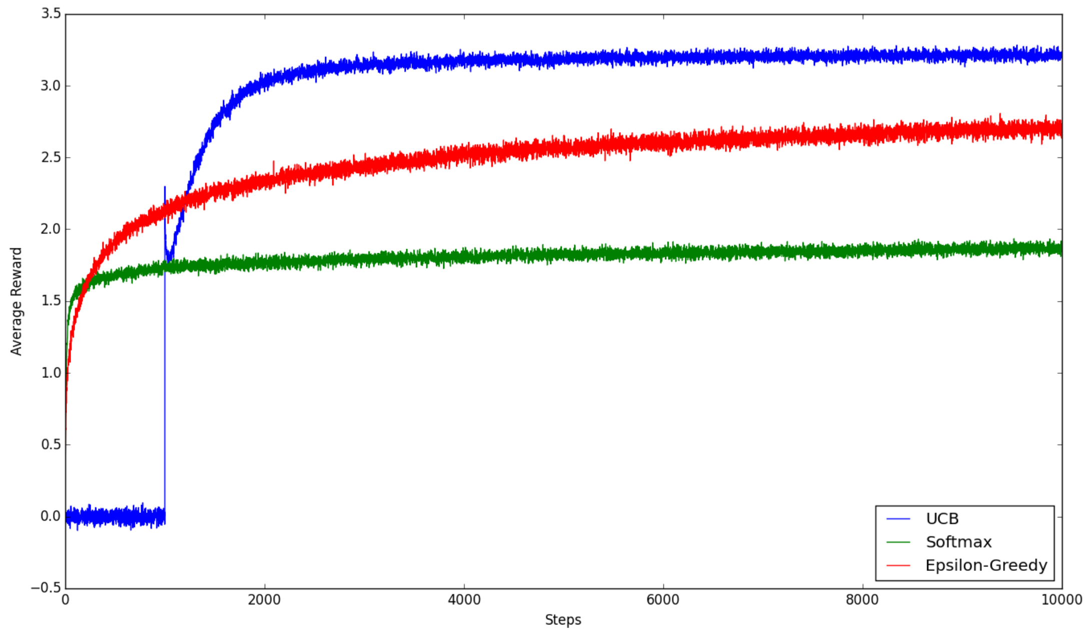

Figure 9: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 10000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

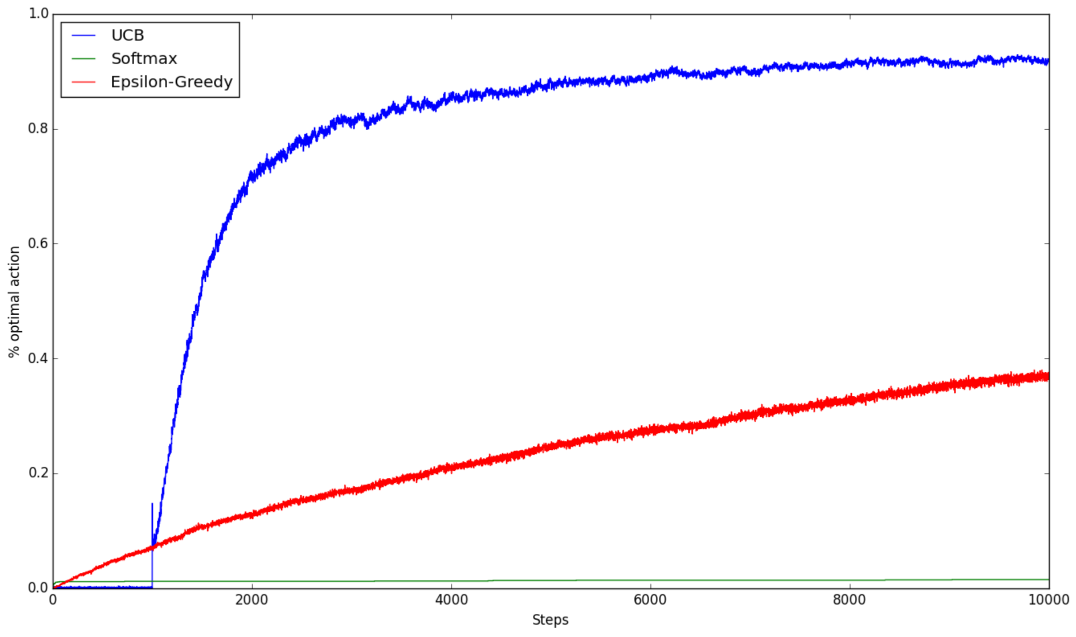

Figure 10: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 10000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

3.4.3 Plots for 20000 time-steps

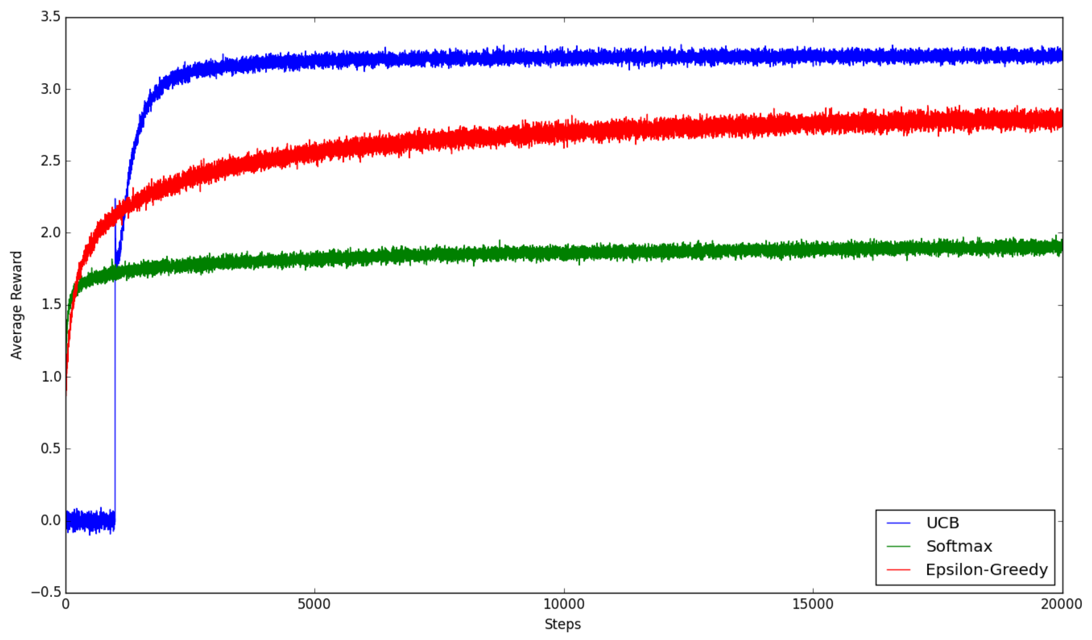

Figure 11: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 20000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

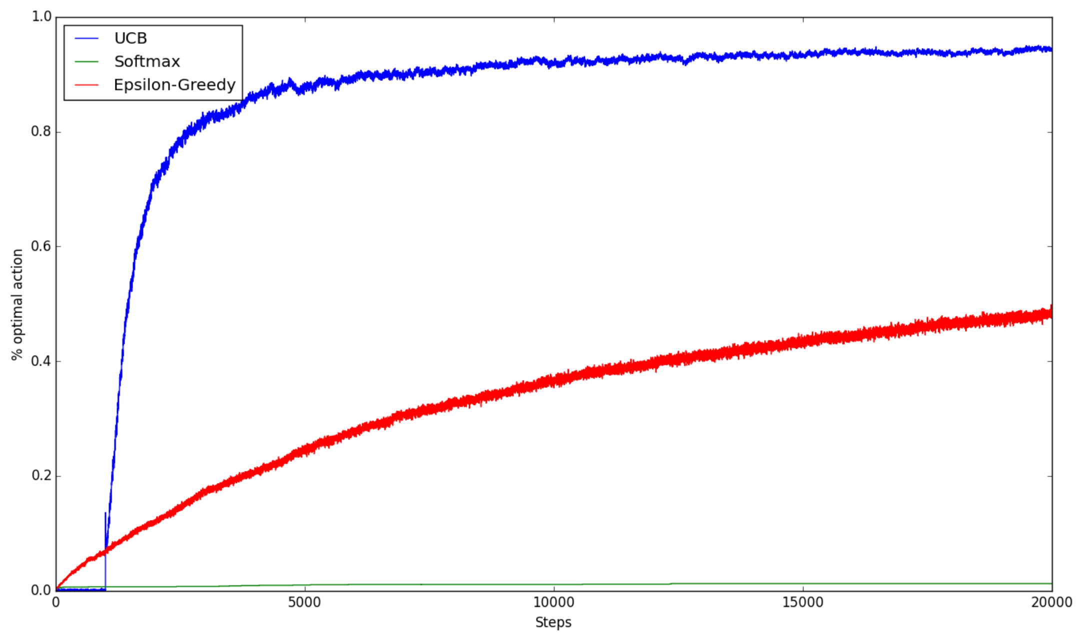

Figure 12: Plot comparing the average reward accrued using UCB, ϵ-greedy and Softmax Action Selection on the 1000-armed testbed over 20000 timesteps. These data are averaged over 2000 runs of different bandit problems. For ϵ-greedy approach, $\epsilon = 0.1$ and for softmax action selection, $\tau = 0.1$ where τ is the temperature parameter.

4 Analysis of the Plots

4.1 Plots for ϵ-greedy action selection for 10 arms

The simulation for ϵ-greedy action selection is run on a 10-armed bandit testbed for three different values of ϵ which are 0.0 (greedy) , 0.1 (being greedy 90 % of the time) , 0.01 (being greedy 99 % of the time). We can notice in the plot of these simulations that over a time-step of 1000, $\epsilon = 0.1$ does the best both in terms of average reward and % optimal action selection. Notice that in the long run $\epsilon = 0.01$ might do better than $\epsilon = 0.1$ both in terms of average reward and % optimal action selection. The greedy method ($\epsilon = 0.0$) performs significantly worse in the long run because it often gets stuck performing a suboptimal action. The % optimal action selection shows that the greedy action selection found the optimal action in only approximately one-third of the tasks. Therefore in two-third of the tasks,its initial samples of optimal action were bad and it never returned to them. In comparison to greedy, ϵ-greedy perform better because they continue to explore and improve their chance of recognizing the optimal action.

4.2 Plots for Softmax Action Selection for 10 arms

The simulation for softmax action selection is run on a 10-armed bandit testbed for four different values of temperature which are 0.01, 0.1, 1 and 10. We can notice in the plot of these simulations that over a time-step of 1000, temperature of 0.1 does the best both in terms of average reward and % optimal action selection. We can notice that if the value of temperature is too low (ie 0.01), we essentially end up behaving greedily but it often gets stuck performing a suboptimal action. Similarly we can also notice that if the value of temperature is too high (ie 10), we essentially end up being random in our action selection and perform badly and hence % optimal action is only 10% ie we are picking up the optimal arm uniformly randomly. We can notice that temperature value of 0.1 is the most apt as it carefully balances exploration and exploitation. Also this is the only temperature which continues to grow for both the plots while the other temperatures seem to saturate out.

4.3 Plots for UCB vs ϵ-greedy vs Softmax Action Selection for 10 arms

The simulation for comparison of UCB with other action selection plots is run on a 10-armed bandit testbed for $\epsilon = 0.1$ for ϵ-greedy and temperature $= 0.1$ for softmax action selection over a 1000 timesteps. We can clearly notice that UCB does the best both in terms of average reward and % optimal action selection. Initially, UCB performs bad but that can be attributed to the fact that initially UCB plays all the onces once and is forced to take suboptimal actions. Also in comparison ϵ-greedy performs better that softmax action selection for this particular choice of ϵ and temperature. But we can conclude that UCB performs the best in case of 10 arms over 1000 timesteps.

4.4 Plots for UCB vs ϵ-greedy vs Softmax Action Selection for 1000 arms

Now we do the comparison of algorithms as the number of arms increase. We increase the number of arms from 10 to 1000 and compare the algorithms for different values of time-steps.

4.4.1 Plots for 1000 timesteps

If we keep the number of timesteps the same as the one in the previous question, we can notice that UCB performs the worst in comparison to ϵ-greedy and softmax action selection. ϵ-greedy performs the best eventually(ie in 1000 timesteps) because it starts picking up sub-optimal action which still does better than UCB because UCB is continuing to play each arm one and hence doesn't repeatedly take a sub-optimal action. In the ϵ-greedy action selection doesn't visit all the arms and selects action from a set of few sampled arms. It is not fair to compare ϵ-greedy and softmax action selection amongst themselves as these are parameterized by ϵ and temperature and hence for different values of ϵ and temperature may yield different results but this is sufficient to conclude that both ϵ-greedy and softmax action selection perform better than UCB for 1000 time-steps simply because UCB continues to play each arm once while ϵ-greedy and softmax action selection behave greedily majority of the time. This can also be seen from the %optimal action plot which shows that UCB selects optimal action once in 1000 times which should be expected. This is also not fair on UCB and hence in the next analysis we increase the time step from 1000 to 10000 and 20000.

4.4.2 Plots for 10000 timesteps

As we increase the number of timesteps from 1000 to 10000 we can immediately see that UCB performs best in comparison to ϵ-greedy and softmax action selection. Initially again, UCB performs the worst but that can again be attributed to the fact that UCB is playing each arm once. We can also see that there is sudden shoot in average reward after 1000 steps which can be attributed to the fact that UCB now starts being greedy with respect to the upper confidence bound on the estimates. Again we see that ϵ-greedy does better than softmax action selection which is not a fair comparison as they are parameterised by ϵ and temperature.

4.4.3 Plots for 20000 timesteps

The results for 20000 timesteps are similar to the ones for 10000 timesteps. One thing to notice though is that ϵ-greedy seems to be converging to UCB in terms of average reward and % optimal action selection.

5 Conclusion

- ϵ-greedy T = 1000 and N = 10
  - maximum average reward is obtained for $\epsilon = 0.1$
  - maximum optimal action is obtained for $\epsilon = 0.1$
- softmax action selection T = 1000 and N = 10
  - maximum average reward is obtained for temperature = 0.1
  - maximum optimal action is obtained for temperature = 0.1
- UCB vs ϵ-greedy vs softmax action selection T = 1000 and N = 10
  - maximum average reward is obtained for UCB
  - maximum optimal action is obtained for UCB
- UCB vs ϵ-greedy vs softmax action selection T = 1000 and N = 1000
  - maximum average reward is obtained for ϵ-greedy with $\epsilon = 0.1$
  - maximum optimal action is obtained for ϵ-greedy with $\epsilon = 0.1$
- UCB vs ϵ-greedy vs softmax action selection T = 10000 and N = 1000
  - maximum average reward is obtained for UCB
  - maximum optimal action is obtained for UCB
- UCB vs ϵ-greedy vs softmax action selection T = 20000 and N = 1000
  - maximum average reward is obtained for UCB
  - maximum optimal action is obtained for UCB

6 Code Listing

6.1 Bandit Class for object instantiation

```python
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

''' Define a Bandit class which has the following parameters
@param numArm : number of actions in the K-armed bandit problem
@param epsilon : Set the epsilon value for epsilon-greedy action selection. Default value is 0.0
@param temperature : Set the temperature parameter of softmax action selection. Default value is 1.0
@param epsilonGreedy : Flag to select that action-selection is based on epsilon-greedy algorithm. Default value is False
@param softmax : Flag to select that action-selection is based on the softmax algorithm. Defalut value is False
@param ucb : Flag to select that action-selection is based on the softmax algorithm. Default value is False
@param initial_estimate : Setting the initial estimates for estimates all the arms
@param trueReward : Setting the true reward for the all arms
'''

class Bandit:
    # Constructor function to initialise the parameters for the bandit instantiation
    def __init__(self, numArm = 10, epsilon = 0.0, temperature = 1.0, epsilonGreedy = False, softmax = False, ucb =
    False, initial_estimate = 0.0, trueReward = 0.0):
        # Initialisation of parameters
        self.numArm = numArm
        self.epsilon = epsilon
        self.temperature = temperature
        self.time = 0
        self.averageReward = 0
        self.epsilonGreedy = epsilonGreedy
        self.softmax = softmax
        self.ucb = ucb
        self.armIndices = np.arange(self.numArm)
        self.qTrue = trueReward * np.ones(self.numArm) # True rewards for each action
        self.qEst = initial_estimate *  np.ones(self.numArm) # Estimate of the reward for each action
        self.actionCount = np.zeros(self.numArm) # Action Selection count
        # Get the true rewards for each action
        for i in range(0,self.numArm):
            self.qTrue[i] = self.qTrue[i] + np.random.randn()
        self.bestAction = np.argmax(self.qTrue)

    # Function to choose the action based on what algorithm is being followed
    def chooseAction(self):
        if self.epsilonGreedy == True:
            armToPlay = np.random.randint(0,self.numArm) # Choose a random action
            # With probability greater than epsilon, choose the action greedily
            if np.random.random() > self.epsilon:
                armToPlay = np.argmax(self.qEst)
            return armToPlay

        if self.softmax == True:
            expo_val = np.exp(self.qEst / self.temperature)
            prob_val = expo_val / np.sum(expo_val) # Get the probability distribution to be sampled from
            rnd = np.random.random() # Sample a random number
            # Use the CDF method to sample from the Gibbs distribution
            for i,w in enumerate(prob_val):
                rnd -= w
                if rnd < 0:
                    return i
            return 0

        if self.ucb == True:
            # play each action once at first
            if(np.count_nonzero(self.actionCount) < self.numArm):
                return np.argmin(self.actionCount)
            # after having played each arm once, choose action greedily using the upper confidence bound
            return np.argmax(self.qEst + np.sqrt(np.log(self.time + 1) / (np.asarray(self.actionCount) + 1)))

    # Function to get the reward on playing the action, increment the count of action and update the estimate of that action
    def getReward(self,action):
        reward = np.random.randn() + self.qTrue[action]
        self.time += 1
        self.averageReward = (self.time - 1.0) / self.time * self.averageReward + reward / self.time # keep a running average of the reward
        self.actionCount[action] += 1 # increase the action count
        self.qEst[action] += 1.0 / self.actionCount[action] * (reward - self.qEst[action]) # update the estimate of the action
        return reward
```

../bandit.py

6.2 Python Library for Simulating the Bandit Problem

```python
# Import the necessary libraries
import numpy as np

''' Function to simulate the bandit problem
@param numDiffBandits : variable stating the number of different bandit problems
@param timeSteps : variable stating the number of time steps over which the actions are taken
@param bandits : list contains the different bandit problems
'''
def simulateBandit(numDiffBandits, timeSteps, bandits):
    optimalActionCount = np.zeros(shape=(len(bandits),timeSteps)) # 2D matrices for storing the count of optimal action selection in each time step for all the different bandit problems
    averageReward = np.zeros(shape=(len(bandits),timeSteps)) # 2D matrices for storing the average reward in each time step for all the different bandit problems
    for banditIndex, bandit in enumerate(bandits): # iterating through the all the different bandit problems
        for i in range(0,numDiffBandits): # iterating through the number of different bandits
            for j in range(0,timeSteps): # iterating through the time steps for each bandit problem
                action = bandit[i].chooseAction() # select an action to take at each time step depending on the action selection method for the particular bandit
                reward = bandit[i].getReward(action) # get a reward for selecting that action for that particular time step
                averageReward[banditIndex][j] += reward # calculate the cummulative reward
                if action == bandit[i].bestAction:
                    optimalActionCount[banditIndex][j] += 1 # increment the count of action by 1
        optimalActionCount[banditIndex] /= numDiffBandits # average out the optimal action count
        averageReward[banditIndex] /= numDiffBandits # average out the cummulative reward
    return optimalActionCount, averageReward
```

../simulateBandit.py

6.3 Python Script to Generate Plots for ϵ-greedy action selection

```python
''' Run this python script to generate the plots for the epsilon-greedy plot on a
    N-armed bandit testbed'''
from bandit import Bandit # import the bandit class to instantiate bandit objects for
    different bandit problems
from simulateBandit import simulateBandit # import the simulateBandit function to do
    the simulation of arm pulls and getting rewards
import matplotlib.pyplot as plt # for plotting graphs

''' Function used for plotting the average reward and optimal action selection using
    epsilon greedy action selection methods
@param numDiffBandits : number of different bandit problems
@param timeSteps : number of different time steps over which the algorithm is run
'''
def epsilonGreedy(numDiffBandits, timeSteps):
    epsilons = [0.0 , 0.1, 0.01] # taking three different values of epsilon
    bandits = list() # making a list of bandit problems
    # iterating through the list of epsilons and adding different bandit problems
    for epsilonIndex, epsilon in enumerate(epsilons):
        bandits.append([Bandit(epsilon = epsilon, epsilonGreedy = True) for _ in
            range(0,numDiffBandits)])
    optimalActionCount, averageReward = simulateBandit(numDiffBandits, timeSteps,
        bandits)
    # plotting the results for optimal action
    plt.figure(0)
    for epsilon, counts in zip(epsilons, optimalActionCount):
        plt.plot(counts, label='epsilon = ' + str(epsilon))
    plt.xlabel('Steps')
    plt.ylabel('% optimal action')
    plt.legend(loc='best')
    # plotting the results for average reward
    plt.figure(1)
    for epsilon, rewards in zip(epsilons, averageReward):
        plt.plot(rewards, label='epsilon = ' + str(epsilon))
    plt.xlabel('Steps')
    plt.ylabel('Average Reward')
    plt.legend(loc='best')

epsilonGreedy(2000,1000) # running the simulation on 10-armed bandit testbed for 2000
    different bandit problems over 1000 time steps
plt.show()
```

../epsilonGreedy.py

6.4 Python Script to Generate Plots for softmax action selection

```python
''' Run this python script to generate the plots for the softmax action selection
    plots on a N-armed bandit testbed'''
from bandit import Bandit # import the bandit class to instantiate bandit objects for
    different bandit problems
from simulateBandit import simulateBandit # import the simulateBandit function to do
    the simulation of arm pulls and getting rewards
import matplotlib.pyplot as plt # for plotting graphs

''' Function used for plotting the average reward and optimal action selection using
    softmax action selection methods
@param numDiffBandits : number of different bandit problems
@param timeSteps : number of different time steps over which the algorithm is run
'''
def softmax(numDiffBandits,timeSteps):
    temperatures = [0.01,0.1,1,10]  # taking 4 different values of temperate for the
        softmax distribution
    bandits = list() # making a list of bandit problems
    # iterating through the list of temperatures and adding different bandit problems
    for tempIndex, temp in enumerate(temperatures):
        bandits.append([Bandit(temperature = temp, softmax = True) for _ in
            range(0,numDiffBandits)])
    optimalActionCount, averageReward = simulateBandit(numDiffBandits, timeSteps,
        bandits)
    # plotting the results for optimal action
    plt.figure(0)
    for temp, counts in zip(temperatures, optimalActionCount):
        plt.plot(counts, label='temp = ' + str(temp))
    plt.xlabel('Steps')
    plt.ylabel('% optimal action')
    plt.legend(loc='best')
    # plotting the results for average reward
    plt.figure(1)
    for temp, rewards in zip(temperatures, averageReward):
        plt.plot(rewards, label='temp = ' + str(temp))
    plt.xlabel('Steps')
    plt.ylabel('Average Reward')
    plt.legend(loc='best')

softmax(2000,1000) # running the simulation on 10-armed bandit testbed for 2000
    different bandit problems over 1000 time steps
plt.show()
```

../softmax.py

6.5 Python Script to Generate Plots for comparing UCB vs ϵ-greedy action selection vs softmax action selection

```python
''' Run this python script to generate the plots for comparing the UCB algorithm vs
    epsilon greedy and softmax on a N-armed bandit testbed'''
from bandit import Bandit # import the bandit class to instantiate bandit objects for
    different bandit problems
from simulateBandit import simulateBandit # import the simulateBandit function to do
    the simulation of arm pulls and getting rewards
import matplotlib.pyplot as plt # for plotting graphs

''' Function used for plotting the average reward and optimal action selection using
    ucb,epsilon greedy and softmax action selection methods
@param numArms : number of arms in the testbed
@param numDiffBandits : number of different bandit problems
@param timeSteps : number of different time steps over which the algorithm is run
'''
def ucb(numArms,numDiffBandits, timeSteps):
    algos = ['UCB','Softmax','Epsilon Greedy'] # iterating through the different
        algorithms
    bandits = [[],[],[]]
    bandits[0] = [Bandit(numArm = numArms, ucb = True) for _ in
        range(0,numDiffBandits)] # list of bandits following ucb
    bandits[1] = [Bandit(numArm = numArms, softmax = True, temperature = 0.1) for _ in
        range(0,numDiffBandits)] # list of bandits following softmax action selection
    bandits[2] = [Bandit(numArm = numArms, epsilonGreedy = True, epsilon = 0.1) for _
        in range(0,numDiffBandits)] # list of bandits following epsilon greedy selection
    optimalActionCount, averageReward = simulateBandit(numDiffBandits, timeSteps,
        bandits)
    # plotting the results for optimal action
    plt.figure(0)
    plt.plot(optimalActionCount[0],label='UCB')
    plt.plot(optimalActionCount[1],label='Softmax')
    plt.plot(optimalActionCount[2],label='Epsilon-Greedy')
    plt.xlabel('Steps')
    plt.ylabel('% optimal action')
    plt.legend(loc = 'best')
    # plotting the results for average reward
    plt.figure(1)
    plt.plot(averageReward[0],label = 'UCB')
    plt.plot(averageReward[1],label = 'Softmax')
    plt.plot(averageReward[2],label = 'Epsilon-Greedy')
    plt.xlabel('Steps')
    plt.ylabel('Average Reward')
    plt.legend(loc = 'best')

ucb(1000,2000,20000) # running the simulation on 1000-armed bandit testbed for 2000
    different bandit problems over 20000 time steps
plt.show()
```

../ucb.py