Assignment 9

July 8, 2018

0.0.1 Team members

Swaroop Bhandary K

Supriya Vadiraj

Vajra Ganeshkumar

```
In [77]: import numpy as np
         import matplotlib.pyplot as plt
```

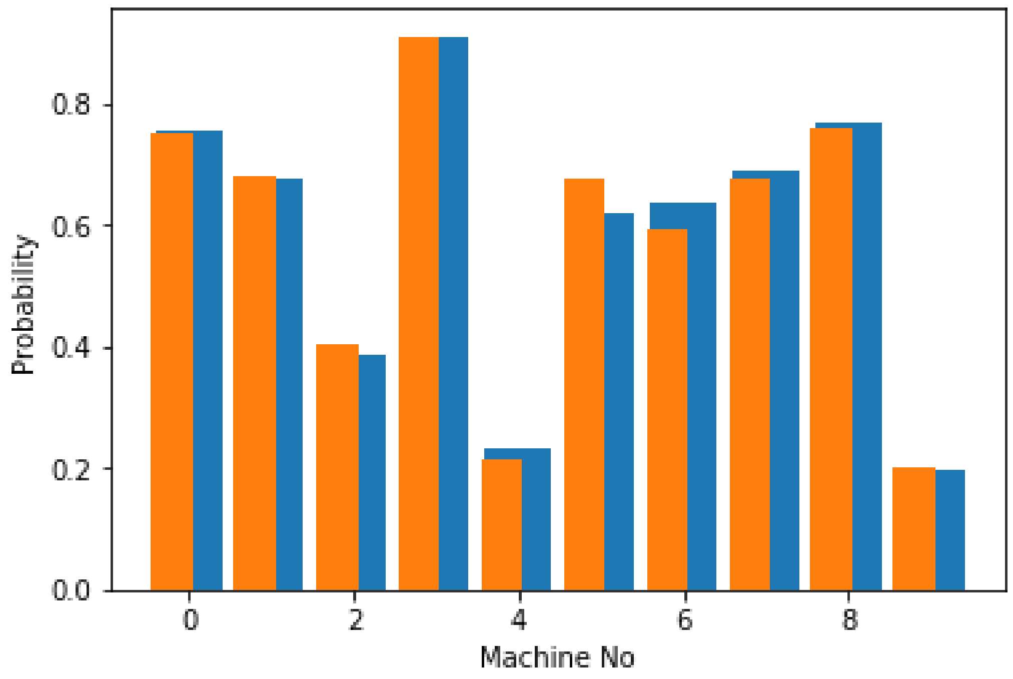
0.1 K bandit implementation.

0.1.1 Epsilon greedy algorithm has been used to find the policy

```
In [23]: class Agent():
             def __init__(self, k, epsilon):
                 self.k = k
                 self.epsilon = epsilon
                 self.times_machine_chosen = np.zeros(k)
                 self.q_value = np.zeros(k)

             def update_qvalue(self, action, value):
                 self.times_machine_chosen[action] +=1
                 self.q_value[action] += (1./self.times_machine_chosen[action]) * (value - self.

             def choose_action(self):
                 rand = np.random.random()
                 if rand < self.epsilon:
                     return np.random.randint(self.k)
                 else:
                     return np.argmax(self.q_value)

         def get_reward(action, p_reward):
             rand = np.random.random()
             if rand < p_reward[action]:
                 return 1
             else:
```

```
            return 0

    def experiment(k, p_reward):
        epsilon = 0.2
        no_of_episodes = 20000

        agent = Agent(k, epsilon)
        no_of_wins_per_machine = np.zeros(k)
        no_of_pulls_per_machine = np.zeros(k)
        action_history = []
        reward_history = []
        for i in range(no_of_episodes):
            action = agent.choose_action()
            reward = get_reward(action, p_reward)
            agent.update_qvalue(action, reward)
            action_history.append(action)
            reward_history.append(reward)

            if reward == 1:
                no_of_wins_per_machine[action] += 1
            no_of_pulls_per_machine[action] +=1

        obtained_prob = [no_of_wins_per_machine[j]/no_of_pulls_per_machine[j] for j in rang
        return obtained_prob, reward_history
```

```
In [16]: k = 10
         p_reward = [np.random.random_sample() for i in range(k)]

         obtained_prob, rewards = experiment(k, p_reward)

         offset = np.full((k,), 0.2)
         plt.bar(range(k), obtained_prob)
         plt.bar(range(k)-offset, p_reward, width=0.5)
         plt.xlabel('Machine No')
         plt.ylabel('Probability')

Out[16]: Text(0,0.5,u'Probability')
```



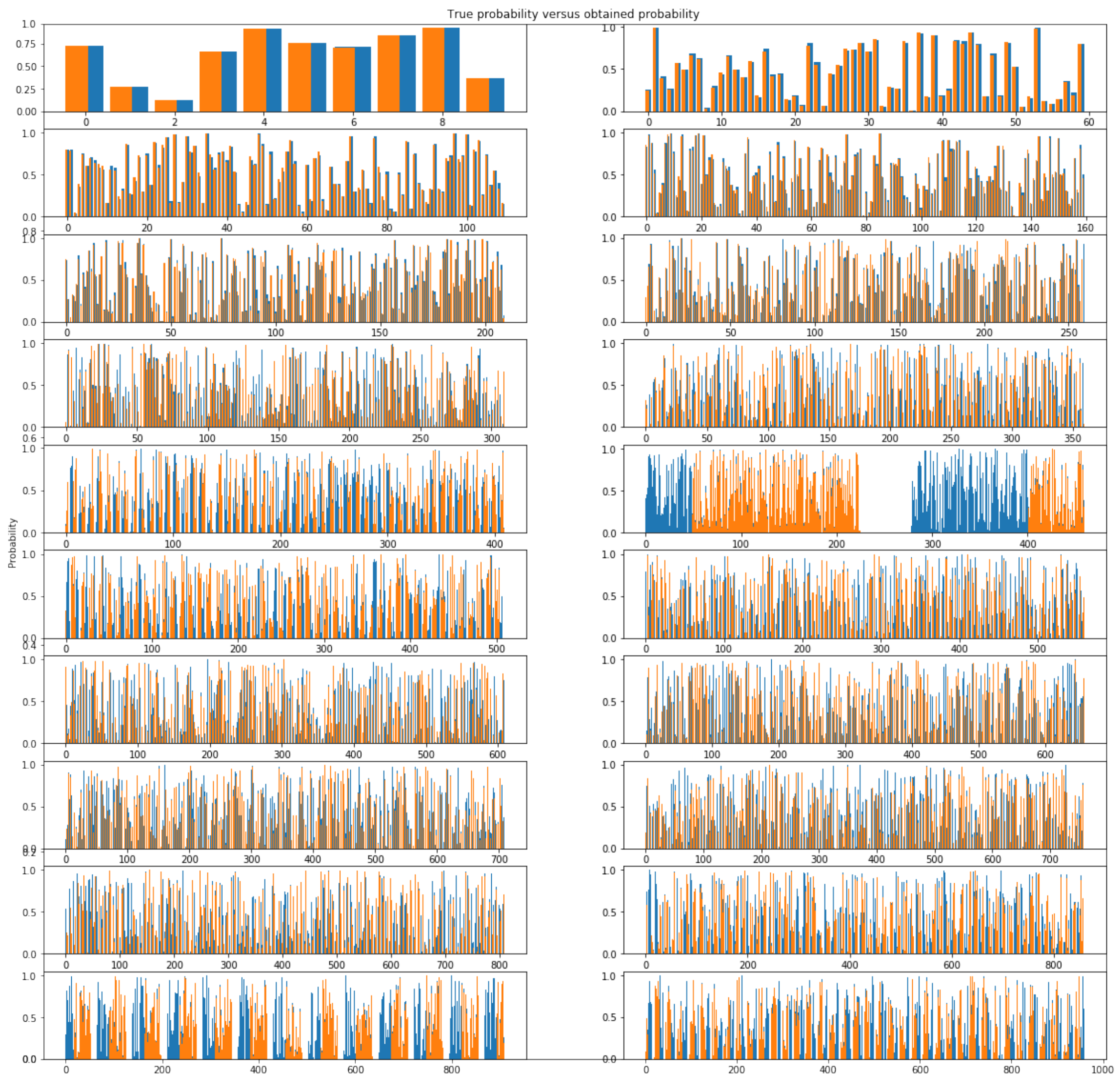
0.2 Test the implementation by varying k value and performing statistical analysis on the obtained results.

0.2.1 Statistical analysis has been performed in 2 ways:

Comparing obtained probability distribution and true probability distribution

Checking if the algorithm is able to find the optimal machine to use

```
In [22]: count_reward = []
         f = plt.figure(figsize=(20,20))
         plt.xlabel('Machine')
         plt.ylabel('Probability')
         plt.title('True probability versus obtained probability ')
         plt.gca().axes.get_xaxis().set_visible(False)
         index = 0
         for k in range(10,1000,50):
             p_reward = [np.random.random_sample() for i in range(k)]
             prob, rewards = experiment(k, p_reward)
             count_reward.append(np.sum(rewards))
             ax = f.add_subplot(10,2,index+1)
             index+=1
             offset = np.full((k,), 0.2)
             ax.bar(range(k), prob)
             ax.bar(range(k)-offset, p_reward, width=0.5)
```



The k-arm bandit problem gets harder as k increases. Even with 200000 iterations the algorithm was able to find optimal solution for k = 160. Upto 410, the algorithm was able to find a suboptimal. For values greater than the algorithm failed to converge. We have used obtained probability distribution as the metric to decide if the algorithm is converged.

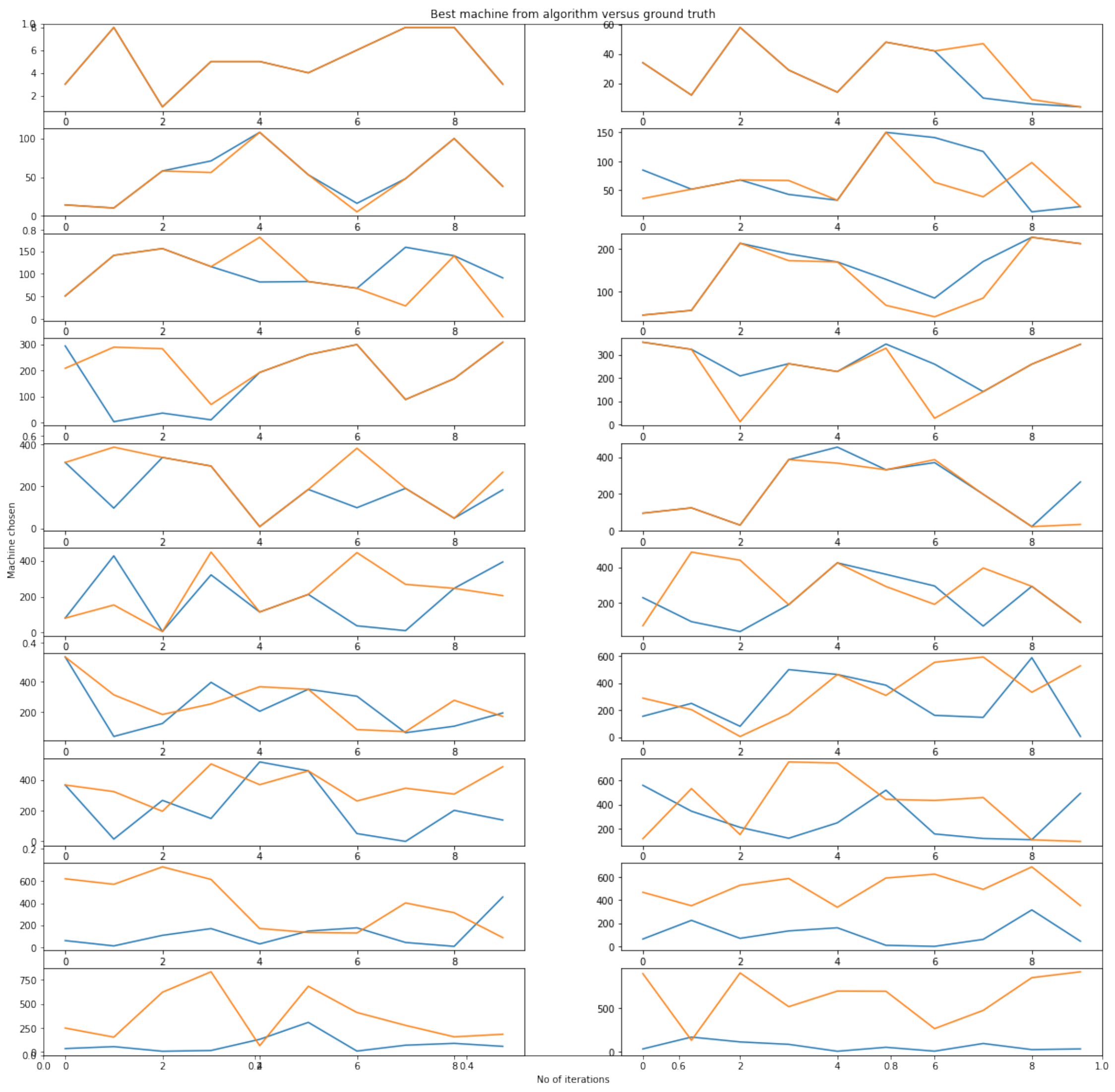
0.2.2 Statistical analysis by using the algorithm to find the best machine. The algorithm is called 10 times for the same value of k to perform statistical analysis

No of episodes have been set to 10000. We perform the statistical analysis by determining if the algorithm is able to find the optimal machine to use.

```
In [25]: num_of_iterations = 10
         epsilon = 0.2
         f = plt.figure(figsize=(20,20))
         plt.title('Best machine from algorithm versus ground truth')
```

```
    plt.xlabel('No of iterations')
    plt.ylabel('Machine chosen')

    index = 0
    for k in range(10,1000,50):
        true_best = []
        obtained_best = []
        for i in range(num_of_iterations):
            p_reward = [np.random.random_sample() for i in range(k)]
            prob, rewards = experiment(k, p_reward)
            true_best.append(np.argmax(p_reward))
            obtained_best.append(np.argmax(prob))
        ax = f.add_subplot(10,2,index+1)
        ax.plot(range(num_of_iterations), obtained_best)
        ax.plot(range(num_of_iterations), true_best)
        index+=1
```

```
/home/swaroop/anaconda2/envs/tf-env/lib/python2.7/site-packages/ipykernel_launcher.py:46: Runtim
```



We can see that the algorithm is able to find the best machine everytime for small values for k. But as the value of k is increased, the algorithm is unable to find the best machine even once.

0.3 Interleaving exploitation and exploration:

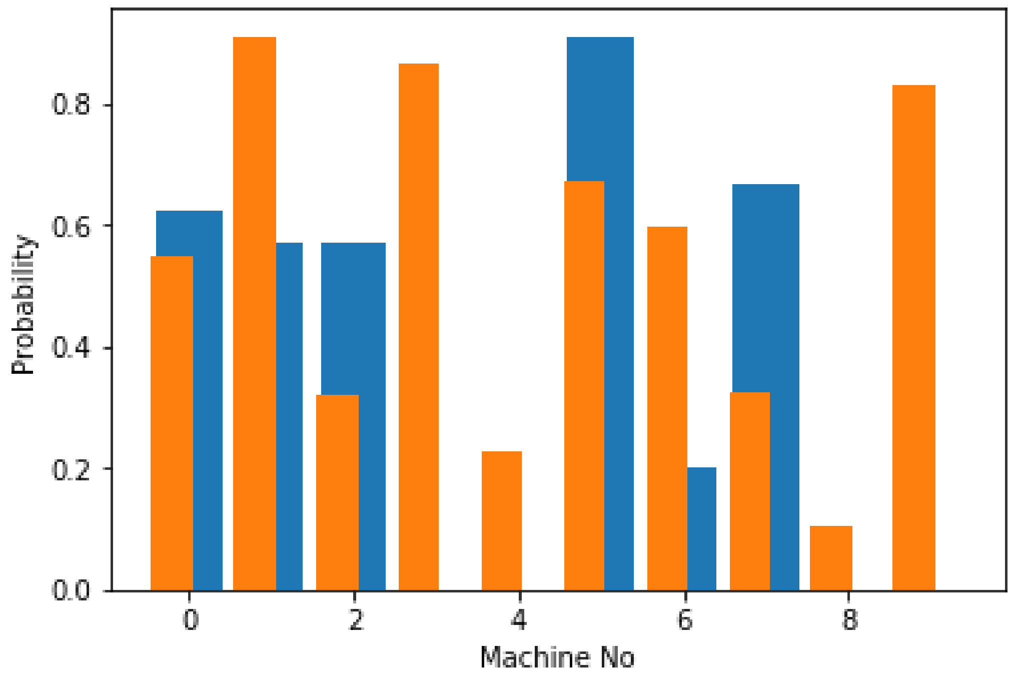
```
In [63]: class Agent():
             def __init__(self, k):
                 self.k = k
                 self.times_machine_chosen = np.zeros(k)
                 self.wins_per_machine = np.zeros(k)
                 self.q_value = np.zeros(k)

             def update_details(self, action, value):
```

```
            self.times_machine_chosen[action] +=1
            self.wins_per_machine[action] +=value
            self.q_value[action] += (1./self.times_machine_chosen[action]) * (value - self.

        def get_reward(self, action, p_reward):
            rand = np.random.random()
            if rand < p_reward[action]:
                return 1
            else:
                return 0
```

```
In [72]: def interleave_search_strategy(k):
             agent1 = Agent(k)
             exp = True
             p_reward = [np.random.random() for i in range(k)]

             for i in range(20000):
                 if exp:
                     action = np.random.randint(k)
                 else:
                     action = np.argmax(agent1.q_value)
                 if i%10 == 0:
                     exp = not exp

                 value = agent1.get_reward(action, p_reward)
                 agent1.update_details(action, value)

             obtained = [agent1.wins_per_machine[j]/agent1.times_machine_chosen[j] if agent1.win
             return obtained
```

```
In [76]: obtained = interleave_search_strategy(160)
         offset = np.full((k,), 0.2)
         plt.bar(range(k), obtained)
         plt.bar(range(k)-offset, p_reward, width=0.5)
         plt.xlabel('Machine No')
         plt.ylabel('Probability')
```

```
Out[76]: Text(0,0.5,u'Probability')
```



We have interleaved exploitation and exploration. Hence the algorithm will first explore the states for 10 episodes, learn the policy and then use this to exploit the structure of the states. After 10 iterations it will explore the states again. //

This will work similar to epsilon greedy approach with epsilon set to 50%. In question 4, we had set epsilon to 20%. Since the amount of exploration is more in question 5, the algorithm is able to perform better when the number of episodes are high. But it's performance degrades when the number of episdoes is less.

0.3.1 Comments on failure cases:

For the epsilon greedy algorithm, we set epsilon value to 20%, hence the algorithm will be exploit only 20% of the times and 80% of the times it will be greedy. Hence if the number of episodes are high, it is better to have a greater value of epsilon. This is because in the long run sticking to a sub optimal policy leads to low rewards. But if the number of episodes is small, it is better to exploit the knowledge that is known instead of exploring and not using the knowledge that is known. //

For values of k from 10 to 160, both epsilon greedy algorithm and interleaving policies algorithm was able to find the optimal solution. But for smaller values of k, epsilon greedy algorithm performed better. //

For values of k from 160 to 400, both epsilon greedy algorithm and interleaving polcies algorithm was able to find suboptimal solutions. But here interleaving policies performed better for the reason mentioned above. //

For values greater than 400, both the algorithm was not able to find the optimal solutions even when we set the number of episodes was increased to 200000. Since the search space is very big.