# INF8953DE Assignment 1

Emile Dimas

TOTAL POINTS

**58.75 / 60**

QUESTION 1

## Question 1: Bandit 30 pts

1.1 Q1.1 **2.75 / 3**

✓ **- 0.25 pts** The implementation is not straightforward/clean.

1.2 Q1.2 **5 / 5**

✓ **- 0 pts** Correct

1.3 Q1.3 **2 / 2**

✓ **- 0 pts** Correct

1.4 Q1.4 **5 / 5**

✓ **- 0 pts** Correct

1.5 Q1.5 **1 / 2**

✓ **- 0.5 pts** partly incorrect comparison/explanation

✓ **- 0.5 pts** partly incorrect plot

1.6 Q1.6 **5 / 5**

✓ **- 0 pts** Correct

1.7 Q1.7 **5 / 5**

✓ **- 0 pts** Correct

1.8 Q1.8 **3 / 3**

✓ **+ 3 pts** Correct

**+ 2 pts** 2/3

**+ 1 pts** 1/3

**+ 1.5 pts** 1.5/3

**+ 2.5 pts** 2.5/3

**+ 0 pts** Click here to replace this description.

QUESTION 2

## Question 2: Dynamic Programming 30 pts

2.1 Q2.1 **8 / 8**

✓ **- 0 pts** Correct

2.2 Q2.2 **4 / 4**

✓ **- 0 pts** Correct

2.3 Q2.3(a) **12 / 12**

✓ **- 0 pts** Correct

2.4 Q2.3(b) **6 / 6**

✓ **- 0 pts** Correct

# Untitled1 (2)

September 29, 2021

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import trange
```

## 1 helper functions

[ ]:
```
def avg_return(arr):
  size = arr.shape
  div = np.repeat(np.expand_dims((np.arange(size[-1]) + 1), 0),size[0], axis=0)
  return np.cumsum(arr, axis =-1)/ div

def total_regret(arr, q_star_a_star):
  size = arr.shape
  div = np.repeat(np.expand_dims((np.arange(size[-1]) + 1), 0),size[0], axis=0)


  b = np.expand_dims(q_star_a_star,1)
  qs = np.repeat(b,size[-1],axis=1)

  optimal = qs * div
  return optimal - np.cumsum(arr, axis =-1)

def compute_VoI(H,Q, eps):
  q_star_a_star = np.array(Q[eps])
  arr = np.array(H[eps])
  reward = avg_return(arr)
  regret = total_regret(arr,q_star_a_star)
  return reward.mean(0), regret.mean(0)
```

## 2 Base Class

[ ]:
```
class MAB:

  def __init__(self, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.nbr_arms = nbr_arms
    self.Q_i = Q_i
```

```
    self.sigma = sigma
    self.mu = mu
    self.q_star = np.random.normal(0, sigma, nbr_arms)
    self.Q = Q_i* np.ones(nbr_arms)
    self.n = np.zeros(nbr_arms)

  def pull(self, action):
    if (action<0) or (action>(self.nbr_arms-1)):
      print('This action is not possible')
      return None
    reward = np.random.normal(self.mu, self.sigma) + self.q_star[action]
    return reward

  def get_q_star_a_star(self):

    return np.max(self.q_star)

  def choose(self, action):
    all_rewards = [self.pull(i) for i in range(self.nbr_arms)]
    best_reward = max(all_rewards)
    actual_reward = all_rewards[action]
    # update
    self.n[action] += 1
    self.Q[action] = (1 - 1.0/self.n[action]) * self.Q[action] + (1.0/self.
↪n[action]) * actual_reward
    # computing regret
    # regret = best_reward - actual_reward
    return actual_reward  #, regret
```

## 3 Epsilon Greedy

[ ]:
```
class EgreedyMAB(MAB):

  def __init__(self, epsilon, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    self.cum_reward = [0]
    # self.total_regret = []
    self.epsilon = epsilon
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def run(self, pulls):
    for pull in range(pulls):
      p = np.random.random()
      if p < self.epsilon:
        action = np.random.choice(self.nbr_arms)
      else:
```

### 1.1 Q1.1 2.75 / 3

✓ **- 0.25 pts** The implementation is not straightforward/clean.

```
    self.sigma = sigma
    self.mu = mu
    self.q_star = np.random.normal(0, sigma, nbr_arms)
    self.Q = Q_i* np.ones(nbr_arms)
    self.n = np.zeros(nbr_arms)

  def pull(self, action):
    if (action<0) or (action>(self.nbr_arms-1)):
      print('This action is not possible')
      return None
    reward = np.random.normal(self.mu, self.sigma) + self.q_star[action]
    return reward

  def get_q_star_a_star(self):

    return np.max(self.q_star)

  def choose(self, action):
    all_rewards = [self.pull(i) for i in range(self.nbr_arms)]
    best_reward = max(all_rewards)
    actual_reward = all_rewards[action]
    # update
    self.n[action] += 1
    self.Q[action] = (1 - 1.0/self.n[action]) * self.Q[action] + (1.0/self.
↪n[action]) * actual_reward
    # computing regret
    # regret = best_reward - actual_reward
    return actual_reward  #, regret
```

## 3 Epsilon Greedy

[ ]:
```
class EgreedyMAB(MAB):

  def __init__(self, epsilon, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    self.cum_reward = [0]
    # self.total_regret = []
    self.epsilon = epsilon
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def run(self, pulls):
    for pull in range(pulls):
      p = np.random.random()
      if p < self.epsilon:
        action = np.random.choice(self.nbr_arms)
      else:
```

```
        action = np.argmax(self.Q)

      actual_reward = self.choose(action)

      # cum_rew = self.cum_reward[-1]/
      # self.cum_reward.append()

      # self.history['action'].append(action)
      self.history.append(actual_reward)
      # self.history['regret'].append(regret)
```

### 3.1 Test

[ ]:
```
epsilons = [0.1, 0.01, 0., 0.9]
runs = 2000
pulls = 1000
H = {}
Q_star = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star = []
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=0, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
    q_star.append(testbed.get_q_star_a_star())
  H[str(epsilon)] = histories
  Q_star[str(epsilon)] = q_star
```

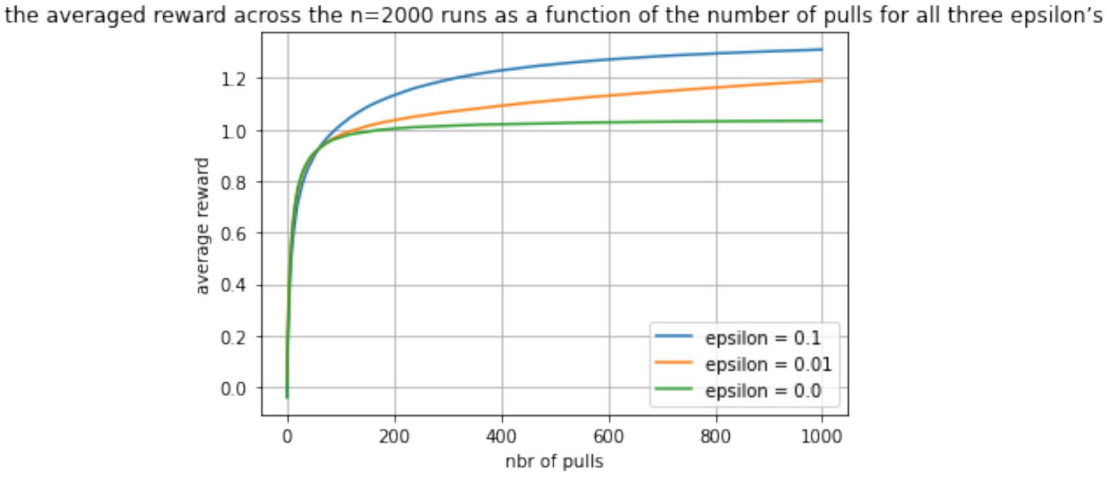
```
------------------------------
Using epsilon = 0.1
100%|      | 2000/2000 [02:43<00:00, 12.26it/s]

------------------------------
Using epsilon = 0.01
100%|      | 2000/2000 [02:47<00:00, 11.90it/s]

------------------------------
Using epsilon = 0.0
100%|      | 2000/2000 [02:46<00:00, 11.98it/s]

------------------------------
Using epsilon = 0.9
100%|      | 2000/2000 [01:55<00:00, 17.33it/s]
```

### 3.2 Presenting the results

[ ]:
```
# epsilon = 0.1
reward_01, regret_01 = compute_VoI(H,Q_star, '0.1')

# epsilon = 0.01
reward_001, regret_001 = compute_VoI(H,Q_star, '0.01')

# epsilon = 0.
reward_00, regret_00 = compute_VoI(H,Q_star, '0.0')
```

[ ]:
```
plt.figure()
plt.title('the averaged reward across the n=2000 runs as a function of the␣
 ↪number of pulls for all three epsilon's')
x = np.arange(len(reward_01))
y1 = reward_01
y2 = reward_001
y3 = reward_00
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['epsilon = 0.1', 'epsilon = 0.01', 'epsilon = 0.0'])
```

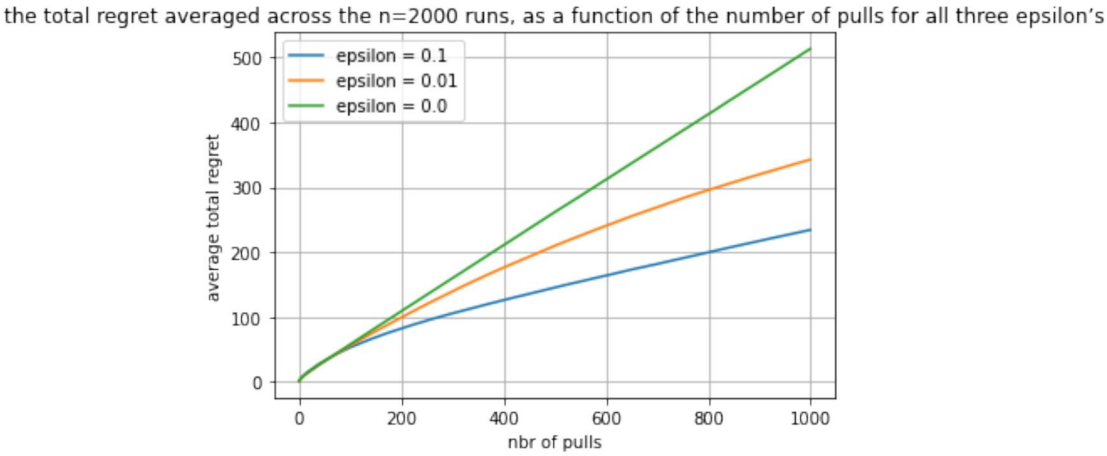
[ ]:
```
<matplotlib.legend.Legend at 0x7f4ccac50ad0>
```




[ ]:
```
plt.figure()
plt.title('the total regret averaged across the n=2000 runs, as a function of␣
 ↪the number of pulls for all three epsilon's')
```

```
x = np.arange(len(regret_01))
y1 = regret_01
y2 = regret_001
y3 = regret_00
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.xlabel('nbr of pulls')
plt.ylabel('average total regret')
plt.grid(True)
plt.legend(['epsilon = 0.1', 'epsilon = 0.01', 'epsilon = 0.0'])
```

[ ]: <matplotlib.legend.Legend at 0x7f4cca6db390>




###Explain what you expected to see and what you actually saw

The 2 figures above show the average cumulative reward and total regret respectively for 1000 steps averaged across 2000 runs. We can see on the reward graph, that the greedy algorithm improved slightly faster than the other methods at the very beginning, but then leveled off at a lower level. In fact, the agent got stuck performing suboptimal actions. The performance can also be seen on the regret graph. We can see that the regret of the greedy algorithm increasing linearly, while the rgegret of the epsilon greedy is sub-linear. This difference in performance can be explained by the fact that the e-greedy agent continue to explore and thus avoid being stuck at a suboptimal action. The degree of exploration can be controlled by the espilon variable. Here we see that the agent that explores more (eps =0.1) performs better in the long run. These results are the same as what I expected

### 1.2 Q1.2 5 / 5

✓ - 0 pts Correct

```
    self.sigma = sigma
    self.mu = mu
    self.q_star = np.random.normal(0, sigma, nbr_arms)
    self.Q = Q_i* np.ones(nbr_arms)
    self.n = np.zeros(nbr_arms)

  def pull(self, action):
    if (action<0) or (action>(self.nbr_arms-1)):
      print('This action is not possible')
      return None
    reward = np.random.normal(self.mu, self.sigma) + self.q_star[action]
    return reward

  def get_q_star_a_star(self):

    return np.max(self.q_star)

  def choose(self, action):
    all_rewards = [self.pull(i) for i in range(self.nbr_arms)]
    best_reward = max(all_rewards)
    actual_reward = all_rewards[action]
    # update
    self.n[action] += 1
    self.Q[action] = (1 - 1.0/self.n[action]) * self.Q[action] + (1.0/self.
↪n[action]) * actual_reward
    # computing regret
    # regret = best_reward - actual_reward
    return actual_reward  #, regret
```

## 3 Epsilon Greedy

[ ]:
```
class EgreedyMAB(MAB):

  def __init__(self, epsilon, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    self.cum_reward = [0]
    # self.total_regret = []
    self.epsilon = epsilon
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def run(self, pulls):
    for pull in range(pulls):
      p = np.random.random()
      if p < self.epsilon:
        action = np.random.choice(self.nbr_arms)
      else:
```

```
        action = np.argmax(self.Q)

      actual_reward = self.choose(action)

      # cum_rew = self.cum_reward[-1]/
      # self.cum_reward.append()

      # self.history['action'].append(action)
      self.history.append(actual_reward)
      # self.history['regret'].append(regret)
```

### 3.1 Test

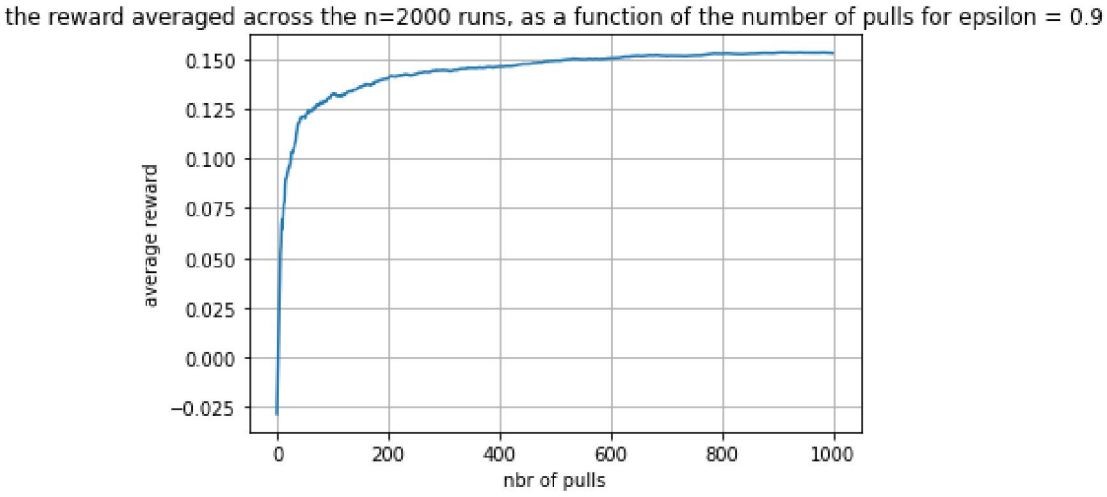
[ ]:
```
epsilons = [0.1, 0.01, 0., 0.9]
runs = 2000
pulls = 1000
H = {}
Q_star = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star = []
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=0, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
    q_star.append(testbed.get_q_star_a_star())
  H[str(epsilon)] = histories
  Q_star[str(epsilon)] = q_star
```

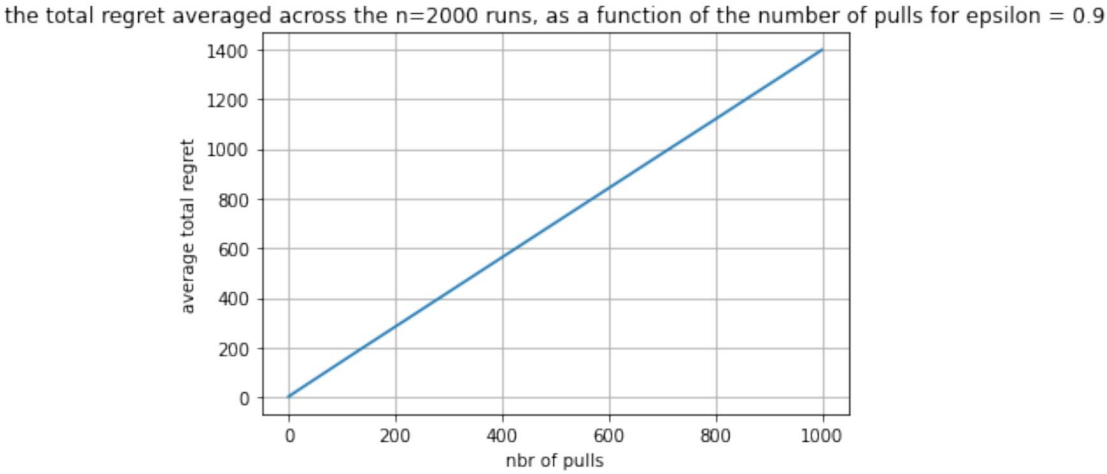
```
------------------------------
Using epsilon = 0.1
100%|      | 2000/2000 [02:43<00:00, 12.26it/s]

------------------------------
Using epsilon = 0.01
100%|      | 2000/2000 [02:47<00:00, 11.90it/s]

------------------------------
Using epsilon = 0.0
100%|      | 2000/2000 [02:46<00:00, 11.98it/s]

------------------------------
Using epsilon = 0.9
100%|      | 2000/2000 [01:55<00:00, 17.33it/s]
```

# 4 Question3)

[ ]:
```
# epsilon = 0.9
reward_09, regret_09 = compute_VoI(H,Q_star, '0.9')
```

[ ]:
```
plt.figure()
plt.title('the reward averaged across the n=2000 runs, as a function of the␣
 ↪number of pulls for epsilon = 0.9')
x = np.arange(len(reward_09))
y1 = reward_09
plt.plot(x, y1)
plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)

plt.figure()
plt.title('the total regret averaged across the n=2000 runs, as a function of␣
 ↪the number of pulls for epsilon = 0.9')
x = np.arange(len(regret_09))
y1 = regret_09
plt.plot(x, y1)
plt.xlabel('nbr of pulls')
plt.ylabel('average total regret')
plt.grid(True)
```






###Do you think setting epsilon = 0.9 is a good strategy? Give reasons why it is or it is not. Plot the average reward and total regret plots for epsilon = 0:9. Explain what you expected to see and what you actually saw.

I don't think using an epsilon of 0.9 is a good idea. This means that 90% of the time the algorithm will take a random action. Even though the problem wasn't stationnary using a too high algorithm would be equivalent to just taking a random action. Depending on the problem, an epsilon that high can be a good thing, but in our case it is not especially that our probelm is stationnary.

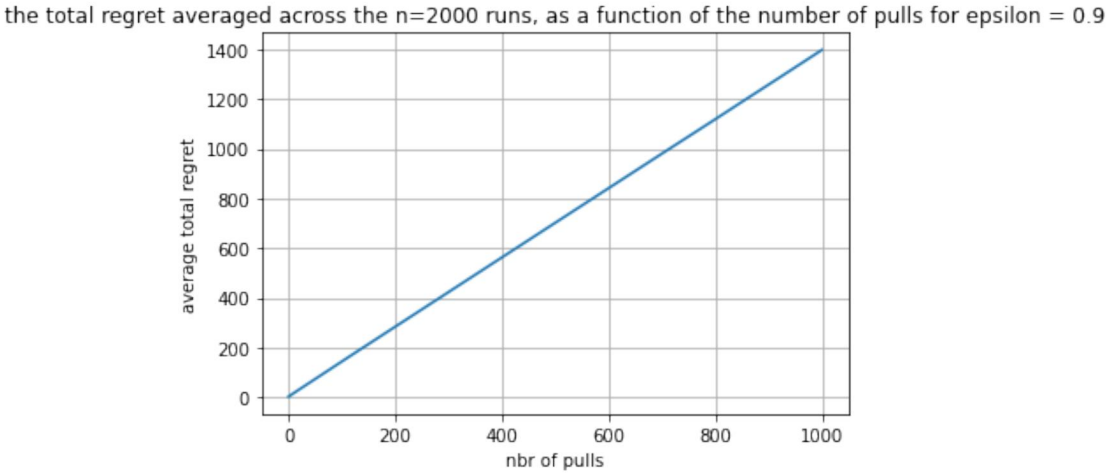
On the graphs, we can see that the agent "converges" to a very subopitmal action. I don't think it really converges since 90% of the time the actions are random, but since we are averaging across 2000 runs, we can see that the agent doesn't get high rewards using this strategy. I was expecting to see a more random reward curve, but since we have 10 actions and 2000 runs, we have the perception that the algorithm converged.

#Q4) Optimistic initial value

[ ]:
```
epsilons = [0.]
runs = 2000
pulls = 1000
H_optimistic = {}
Q_star_optimistic = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star =[]
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=3, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
```

### 1.3 Q1.3 2 / 2

✓ - 0 pts Correct




###Do you think setting epsilon = 0.9 is a good strategy? Give reasons why it is or it is not. Plot the average reward and total regret plots for epsilon = 0:9. Explain what you expected to see and what you actually saw.

I don't think using an epsilon of 0.9 is a good idea. This means that 90% of the time the algorithm will take a random action. Even though the problem wasn't stationnary using a too high algorithm would be equivalent to just taking a random action. Depending on the problem, an epsilon that high can be a good thing, but in our case it is not especially that our probelm is stationnary.

On the graphs, we can see that the agent "converges" to a very subopitmal action. I don't think it really converges since 90% of the time the actions are random, but since we are averaging across 2000 runs, we can see that the agent doesn't get high rewards using this strategy. I was expecting to see a more random reward curve, but since we have 10 actions and 2000 runs, we have the perception that the algorithm converged.

#Q4) Optimistic initial value

[ ]:
```
epsilons = [0.]
runs = 2000
pulls = 1000
H_optimistic = {}
Q_star_optimistic = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star =[]
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=3, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
```

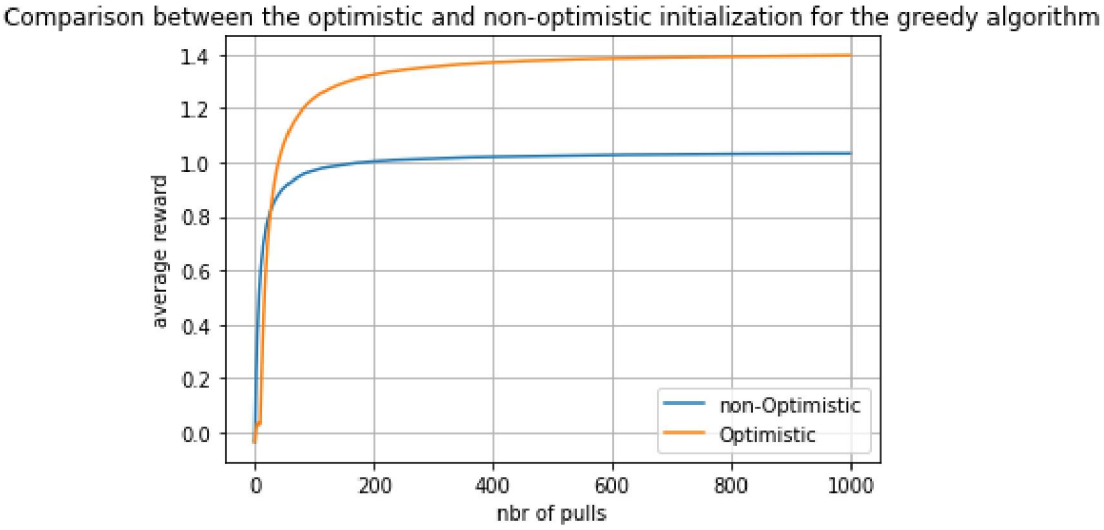
```
    q_star.append(testbed.get_q_star_a_star())
  Q_star_optimistic[str(epsilon)] = q_star
  H_optimistic[str(epsilon)] = histories
```

```
------------------------------
Using epsilon = 0.0
100%|      | 2000/2000 [02:45<00:00, 12.08it/s]
```

[ ]:
```
reward_Optimistic_00, _ = compute_VoI(H_optimistic,Q_star_optimistic, '0.0')
```

[ ]:
```
plt.figure()
plt.title('Comparison between the optimistic and non-optimistic initialization␣
 ↪for the greedy algorithm')
x = np.arange(len(reward_00))
y1 = reward_00
y2 = reward_Optimistic_00

plt.plot(x, y1)
plt.plot(x, y2)

plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['non-Optimistic', 'Optimistic'])
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f4cc6728c50>
```




###Explain what you expected to see and what you actually saw

The graph above shows the average reward for 2 greedy algorithms using different initial values

for the Q estimates. This figure looks like the figure showed previously that illustrates the greedy algorithm vs the epsilon greedy algorithm. In fact, the initial value acts like the epsilon meaning that it makes the agent explore the environmnet. However, unlike the epsilon greedy the optimistic initialisation explores in the begining and the effect of the initialized Q disappear. We can see that the optimistic algo performs way better than when Q_i =0 (since there is no exploration here). Results are like my expectations

## 5 Question 5:

[ ]:
```
epsilons = [0.]
runs = 2000
pulls = 1000
H_Veryoptimistic = {}
Q_star_Veryoptimistic = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star = []
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=100, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
    q_star.append(testbed.get_q_star_a_star())
  H_Veryoptimistic[str(epsilon)] = histories
  Q_star_Veryoptimistic[str(epsilon)] = q_star
```

```
------------------------------
Using epsilon = 0.0
```

```
100%|      | 2000/2000 [02:47<00:00, 11.92it/s]
```

[ ]:
```
reward_VeryOptimistic_00, _ = compute_VoI(H_Veryoptimistic,␣
↪Q_star_Veryoptimistic, '0.0')
```

[ ]:
```
plt.figure()
plt.title('Comparison between the optimistic and non-optimistic initialization␣
↪for the greedy algorithm')
x = np.arange(len(reward_00))
y1 = reward_00
y2 = reward_Optimistic_00
y3 = reward_VeryOptimistic_00
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)

plt.xlabel('nbr of pulls')
```

### 1.4 Q1.4 5 / 5

✓ - 0 pts Correct

for the Q estimates. This figure looks like the figure showed previously that illustrates the greedy algorithm vs the epsilon greedy algorithm. In fact, the initial value acts like the epsilon meaning that it makes the agent explore the environmnet. However, unlike the epsilon greedy the optimistic initialisation explores in the begining and the effect of the initialized Q disappear. We can see that the optimistic algo performs way better than when Q_i =0 (since there is no exploration here). Results are like my expectations

## 5 Question 5:

[ ]:
```
epsilons = [0.]
runs = 2000
pulls = 1000
H_Veryoptimistic = {}
Q_star_Veryoptimistic = {}
for epsilon in epsilons:
  print('_'*30)
  print('Using epsilon = {}'.format(epsilon))
  histories = []
  q_star = []
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = EgreedyMAB(epsilon, nbr_arms=10, Q_i=100, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
    q_star.append(testbed.get_q_star_a_star())
  H_Veryoptimistic[str(epsilon)] = histories
  Q_star_Veryoptimistic[str(epsilon)] = q_star
```

```
------------------------------
Using epsilon = 0.0
```

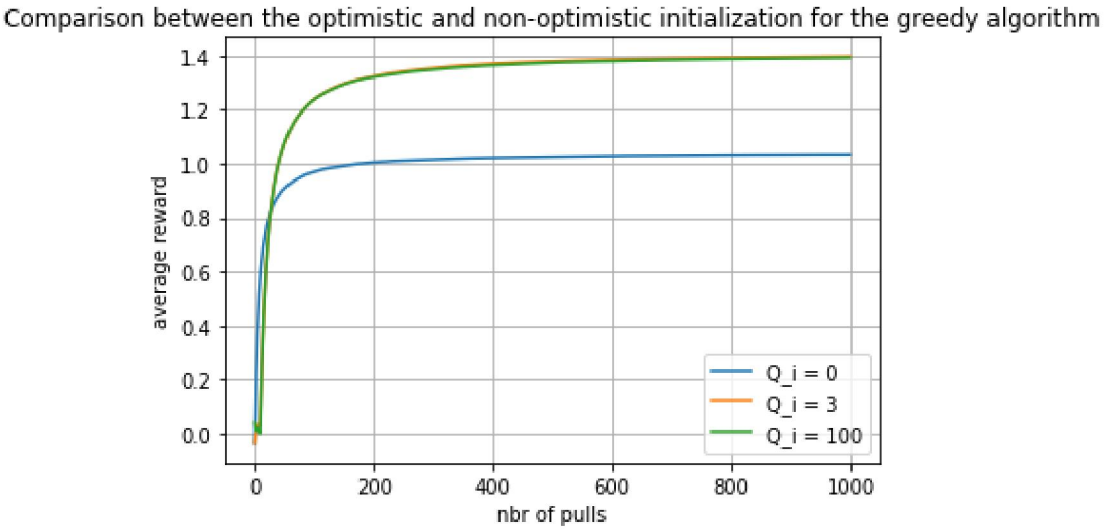
```
100%|      | 2000/2000 [02:47<00:00, 11.92it/s]
```

[ ]:
```
reward_VeryOptimistic_00, _ = compute_VoI(H_Veryoptimistic,␣
↪Q_star_Veryoptimistic, '0.0')
```

[ ]:
```
plt.figure()
plt.title('Comparison between the optimistic and non-optimistic initialization␣
↪for the greedy algorithm')
x = np.arange(len(reward_00))
y1 = reward_00
y2 = reward_Optimistic_00
y3 = reward_VeryOptimistic_00
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)

plt.xlabel('nbr of pulls')
```

```
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['Q_i = 0', 'Q_i = 3', 'Q_i = 100' ])
```

[ ]: <matplotlib.legend.Legend at 0x7f4cc2870790>




###Explain what you expected to see and what you actually saw

This graph is like the pervious one, but we added the results for the agent when Q_i=100. This means higher exploration in the beginning and as expected more exploration in the beginning yields to higher results.

## 6 Question6: UCB

[ ]:
```
class UCBMAB(MAB):

  def __init__(self, c, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    self.c = c
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def run(self, pulls):
    for pull in range(pulls):

      action = np.argmax(self.Q + self.c *np.sqrt(np.log(pull+1)/(self.n+1)))

      actual_reward = self.choose(action)

      self.history.append(actual_reward)
```

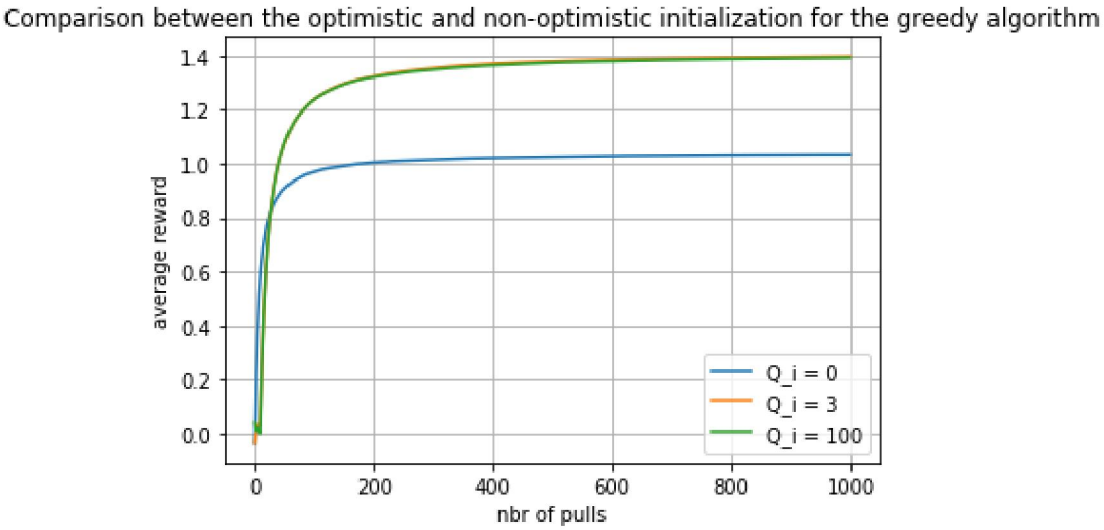
### 1.5 Q1.5 1 / 2

✓ - 0.5 pts partly incorrect comparison/explanation

✓ - 0.5 pts partly incorrect plot

```
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['Q_i = 0', 'Q_i = 3', 'Q_i = 100' ])
```

[ ]: <matplotlib.legend.Legend at 0x7f4cc2870790>




###Explain what you expected to see and what you actually saw

This graph is like the pervious one, but we added the results for the agent when Q_i=100. This means higher exploration in the beginning and as expected more exploration in the beginning yields to higher results.

## 6 Question6: UCB

[ ]:
```
class UCBMAB(MAB):

  def __init__(self, c, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    self.c = c
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def run(self, pulls):
    for pull in range(pulls):

      action = np.argmax(self.Q + self.c *np.sqrt(np.log(pull+1)/(self.n+1)))

      actual_reward = self.choose(action)

      self.history.append(actual_reward)
```

[ ]:
```
K = [0.2, 1, 5.]
runs = 2000
pulls = 1000
H_UCB = {}
Q_star_UCB = {}
for c in K:
  print('_'*30)
  print('Using c = {}'.format(c))
  histories = []
  q_star = []
  for run in trange(runs):
    # print('\tRunning epoch #{}'.format(run+1))
    testbed = UCBMAB(c, nbr_arms=10, Q_i=0, mu=0, sigma=1)
    testbed.run(pulls)
    histories.append(testbed.history)
    q_star.append(testbed.get_q_star_a_star())
  H_UCB[str(c)] = histories
  Q_star_UCB[str(c)] = q_star
```

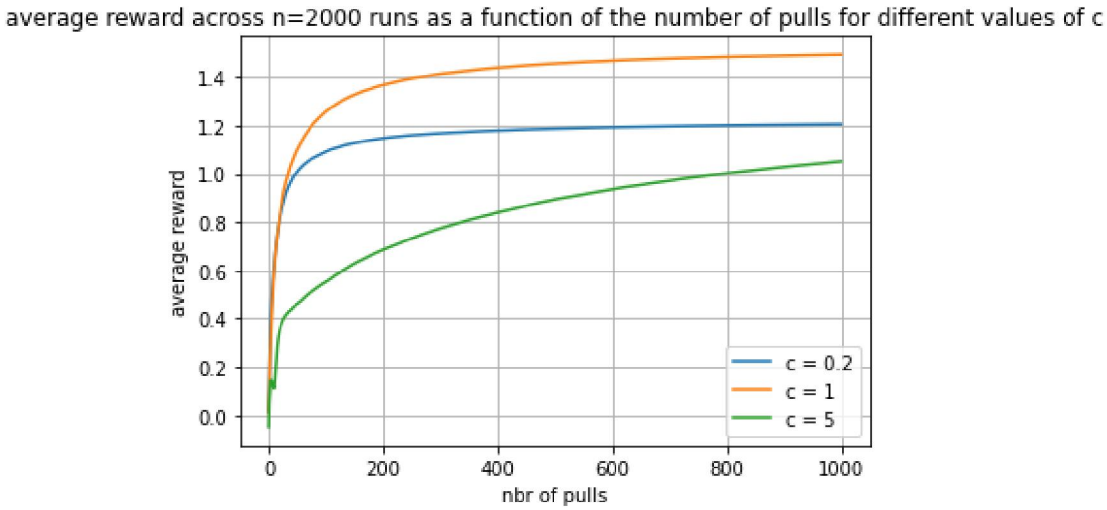
```
------------------------------
Using c = 0.2
100%|      | 2000/2000 [03:06<00:00, 10.75it/s]

------------------------------
Using c = 1
100%|      | 2000/2000 [03:06<00:00, 10.74it/s]

------------------------------
Using c = 5.0
100%|      | 2000/2000 [03:07<00:00, 10.69it/s]
```

[ ]:
```
# c = 0.2
average_reward_02, _ = compute_VoI(H_UCB,Q_star_UCB, '0.2')

# c = 1
average_reward_1, _ =  compute_VoI(H_UCB,Q_star_UCB, '1')

# c = 5
average_reward_5, _ =  compute_VoI(H_UCB,Q_star_UCB, '5.0')
```

[ ]:
```
plt.figure()
plt.title('average reward across n=2000 runs as a function of the number of␣
 ↪pulls for different values of c')
x = np.arange(len(average_reward_02))
y1 = average_reward_02
y2 = average_reward_1
```

```
y3 = average_reward_5
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['c = 0.2', 'c = 1', 'c = 5'])
```

[ ]: <matplotlib.legend.Legend at 0x7f4cb67b9c90>




###Explain what you expected to see and what you actually saw

The UCB algorithm is also based on the paradigm of exploration. Exploration is done by building a confidence interval for the Q estimates. The interval range is controlled by the c hyperparamter. A higher c means a more uncertain estimate, while a lower c means a more certain estimate. The graph above represents the average reward across n=2000 runs across 1000 pulls for 3 values of c (0.2, 1 and 5). We can see that the best results are obtained for c=1. Indeed, a c too small (0.2) will not be able to explore teh different actions, while a c too big will case the agent to be uncertain of all actions even after a lot of experience. I was expecting to see what is showing, except that I wasn't too sure what was going to happen with a c=5

## 7 Gradient Bandit

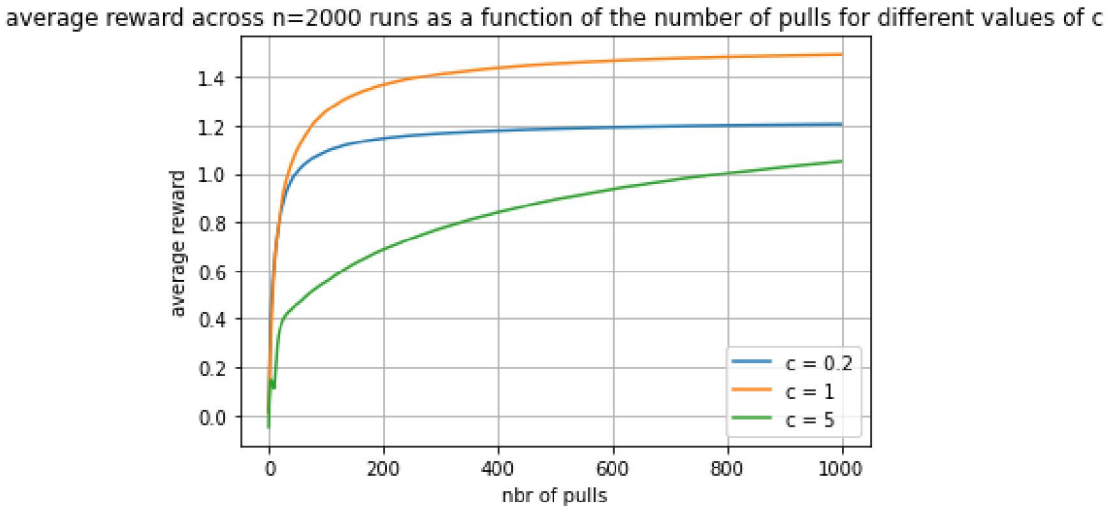
[ ]:
```
def softmax(arr):
  numerator = np.exp(arr) # - max(arr))
  return numerator / np.sum (numerator)
```

### 1.6 Q1.6 5 / 5

✓ - 0 pts Correct

```
y3 = average_reward_5
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['c = 0.2', 'c = 1', 'c = 5'])
```

[ ]: <matplotlib.legend.Legend at 0x7f4cb67b9c90>




###Explain what you expected to see and what you actually saw

The UCB algorithm is also based on the paradigm of exploration. Exploration is done by building a confidence interval for the Q estimates. The interval range is controlled by the c hyperparamter. A higher c means a more uncertain estimate, while a lower c means a more certain estimate. The graph above represents the average reward across n=2000 runs across 1000 pulls for 3 values of c (0.2, 1 and 5). We can see that the best results are obtained for c=1. Indeed, a c too small (0.2) will not be able to explore teh different actions, while a c too big will case the agent to be uncertain of all actions even after a lot of experience. I was expecting to see what is showing, except that I wasn't too sure what was going to happen with a c=5

## 7 Gradient Bandit

[ ]:
```
def softmax(arr):
  numerator = np.exp(arr) # - max(arr))
  return numerator / np.sum (numerator)
```

[ ]:
```
class GB_MAB(MAB):

  def __init__(self, alpha, reward_baseline, nbr_arms=10, Q_i=0, mu=0, sigma=1):
    self.history = []
    # We will use Q as the preference notation
    self.alpha = alpha
    self.reward_baseline = reward_baseline
    super().__init__(nbr_arms, Q_i, mu, sigma)

  def compute_baseline(self):
    if self.reward_baseline==True:
      baseline = np.mean(self.history)
      if len(self.history)!=0:
        return baseline
    return 0

  # overwriting choose function
  def choose(self, action):
    all_rewards = [self.pull(i) for i in range(self.nbr_arms)]
    mask = np.zeros(self.nbr_arms)
    mask[action] = 1.

    # computing probabilities
    probs = softmax(self.Q)


    #computing baseline
    baseline = self.compute_baseline()

    reward = all_rewards[action]

    self.Q = self.Q + self.alpha * (reward - baseline) * (mask - probs)

    # computing regret

    return reward

  def run(self, pulls):
    for pull in range(pulls):

      # choose an action:
      probs = softmax(self.Q)
      action = np.argmax(probs)

      actual_reward = self.choose(action)
```

```
        self.history.append(actual_reward)
```

[ ]:
```
H_GB = {}
Q_star_GB = {}
for baseline in [False, True]:
  print('Using Baseline: {}'.format(baseline))
  for alpha in [0.1, 0.5]:
    print('\t using alpha = {}'.format(alpha))
    histories = []
    q_star = []
    for run in trange(runs):
      #print('\t\tRunning epoch #{}'.format(run+1))
      testbed = GB_MAB(alpha=alpha, reward_baseline=baseline, nbr_arms=10,␣
 ↪Q_i=0, mu=0, sigma=1)
      testbed.run(pulls)

      histories.append(testbed.history)
      q_star.append(testbed.get_q_star_a_star())
    H_GB[str(alpha) + '_' + str(baseline)] = histories
    Q_star_GB[str(alpha) + '_' + str(baseline)] = q_star
```

```
Using Baseline: False
         using alpha = 0.1
100%|      | 2000/2000 [03:36<00:00,  9.22it/s]
         using alpha = 0.5
100%|      | 2000/2000 [03:40<00:00,  9.06it/s]
Using Baseline: True
         using alpha = 0.1
  0%|          | 0/2000 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-
packages/numpy/core/fromnumeric.py:3373: RuntimeWarning: Mean of empty slice.
  out=out, **kwargs)
/usr/local/lib/python3.7/dist-packages/numpy/core/_methods.py:170:
RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
100%|      | 2000/2000 [06:01<00:00,  5.53it/s]
         using alpha = 0.5
100%|      | 2000/2000 [05:53<00:00,  5.65it/s]
```

[ ]:
```
H_GB.keys()
```

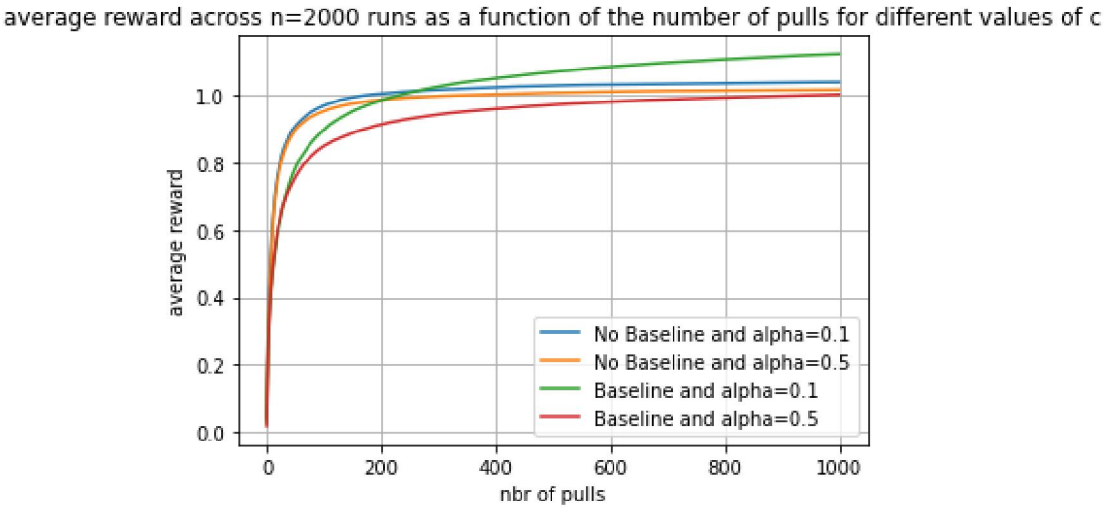
[ ]:
```
dict_keys(['0.1_False', '0.5_False', '0.1_True', '0.5_True'])
```

[ ]:
```
average_reward_GB_NoBaseline_0_1, _ = compute_VoI(H_GB,Q_star_GB, '0.1_False')
average_reward_GB_NoBaseline_0_5, _ = compute_VoI(H_GB,Q_star_GB, '0.5_False')
```

```
average_reward_GB_Baseline_0_1, _ = compute_VoI(H_GB,Q_star_GB, '0.1_True')
average_reward_GB_Baseline_0_5, _ = compute_VoI(H_GB,Q_star_GB, '0.5_True')
```

[ ]:
```
plt.figure()
plt.title('average reward across n=2000 runs as a function of the number of␣
 ↪pulls for different values of c')
x = np.arange(len(average_reward_GB_NoBaseline_0_1))
y1 = average_reward_GB_NoBaseline_0_1
y2 = average_reward_GB_NoBaseline_0_5
y3 = average_reward_GB_Baseline_0_1
y4 = average_reward_GB_Baseline_0_5
plt.plot(x, y1)
plt.plot(x, y2)
plt.plot(x, y3)
plt.plot(x, y4)
plt.xlabel('nbr of pulls')
plt.ylabel('average reward')
plt.grid(True)
plt.legend(['No Baseline and alpha=0.1', 'No Baseline and alpha=0.5', 'Baseline␣
 ↪and alpha=0.1', 'Baseline and alpha=0.5'])
```

[ ]:
```
<matplotlib.legend.Legend at 0x7f4cb678cb90>
```




###Explain what you expected to see and what you actually saw

The figure above displays 4 diiferent curves of average reward across 2000 runs. These setups are a combination of using baseline or not using it and of using a value of alpha equal to 0.1 and 0.5.

We can quickly see that using a baseline improves the performance of the algorithm. Secondly, we can realize that the value of alpha alone cannot help to determine our expectations. While a value of alpha = to 0.5 is better when using no baseline, a value of alpha = 0.1 is better when we use

the baseline. Alpha is the stepsize, so bigger alpha will help the algorithm converge faster, which can be seen in the figure (for the same alpha and comparing baseline vs no).

I wasn't expectinng that the baseline makes a so big difference in the results but as we can see, using baeline helps a lot.

##Question 8) How would you compare the performance of all the 4 different methods: epsilon-greedy, optimistic initial value, UCB, and gradient bandits? How would you rank them in terms of regret?

Obviously, there is no single algorithm that is superior to all the other in all cases. In other words, there is no free lunch. However, based on our experiments, we can compare the different algorithms. these 4 algos have a similarity in the sense that they try to explore the different actions. This exploration is done in different way by different algorithm. For example, epslion-greedy (standard version) has a constant epsilon probability to explore the environment, while the optimistic initial value algo only explores in the beginning. the effect of the initial value disappear across time. in this sense, if the environment is non-stationnary, epsilon greedy would be a better alternative. UCB has also a quasi-constant exploration, however, the more the agent acts in the environmnet the more the confidence interval will shrink and the agent will be more certain of what action to take. Gradient bandit uses an explicit soft policy and samples the action based on the probability which is corrected based in the experience.

Based on our experiments we can rank the different algorithm in terms of regret in this way (best to worse): - UCB - greedy with optimistic initial value - gradient bandit - epsilon-greedy

#---

## 8 Exercise #2

[ ]:
```
# https://github.com/zafarali/emdp#grid-world
! git clone https://github.com/zafarali/emdp.git
```

```
Cloning into 'emdp'…
remote: Enumerating objects: 285, done.
remote: Counting objects: 100% (125/125), done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 285 (delta 76), reused 65 (delta 32), pack-reused 160
Receiving objects: 100% (285/285), 74.78 KiB | 1.31 MiB/s, done.
Resolving deltas: 100% (155/155), done.
```

[ ]:
```
%cd /content/emdp
```

```
/content/emdp
```

[ ]:
```
!pip install -e .
```

```
Obtaining file:///content/emdp
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.7/dist-
packages (from emdp==0.0.5) (1.19.5)
Installing collected packages: emdp
```

### 1.7 Q1.7 5 / 5

✓ - 0 pts Correct

the baseline. Alpha is the stepsize, so bigger alpha will help the algorithm converge faster, which can be seen in the figure (for the same alpha and comparing baseline vs no).

I wasn't expectinng that the baseline makes a so big difference in the results but as we can see, using baeline helps a lot.

##Question 8) How would you compare the performance of all the 4 different methods: epsilon-greedy, optimistic initial value, UCB, and gradient bandits? How would you rank them in terms of regret?

Obviously, there is no single algorithm that is superior to all the other in all cases. In other words, there is no free lunch. However, based on our experiments, we can compare the different algorithms. these 4 algos have a similarity in the sense that they try to explore the different actions. This exploration is done in different way by different algorithm. For example, epslion-greedy (standard version) has a constant epsilon probability to explore the environment, while the optimistic initial value algo only explores in the beginning. the effect of the initial value disappear across time. in this sense, if the environment is non-stationnary, epsilon greedy would be a better alternative. UCB has also a quasi-constant exploration, however, the more the agent acts in the environmnet the more the confidence interval will shrink and the agent will be more certain of what action to take. Gradient bandit uses an explicit soft policy and samples the action based on the probability which is corrected based in the experience.

Based on our experiments we can rank the different algorithm in terms of regret in this way (best to worse): - UCB - greedy with optimistic initial value - gradient bandit - epsilon-greedy

#---

## 8 Exercise #2

[ ]:
```
# https://github.com/zafarali/emdp#grid-world
! git clone https://github.com/zafarali/emdp.git
```

```
Cloning into 'emdp'…
remote: Enumerating objects: 285, done.
remote: Counting objects: 100% (125/125), done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 285 (delta 76), reused 65 (delta 32), pack-reused 160
Receiving objects: 100% (285/285), 74.78 KiB | 1.31 MiB/s, done.
Resolving deltas: 100% (155/155), done.
```

[ ]:
```
%cd /content/emdp
```

```
/content/emdp
```

[ ]:
```
!pip install -e .
```

```
Obtaining file:///content/emdp
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.7/dist-
packages (from emdp==0.0.5) (1.19.5)
Installing collected packages: emdp
```

### 1.8 Q1.8 3 / 3

✓ **+ 3 pts** **Correct**

**+ 2 pts** 2/3

**+ 1 pts** 1/3

**+ 1.5 pts** 1.5/3

**+ 2.5 pts** 2.5/3

**+ 0 pts** Click here to replace this description.

the baseline. Alpha is the stepsize, so bigger alpha will help the algorithm converge faster, which can be seen in the figure (for the same alpha and comparing baseline vs no).

I wasn't expectinng that the baseline makes a so big difference in the results but as we can see, using baeline helps a lot.

##Question 8) How would you compare the performance of all the 4 different methods: epsilon-greedy, optimistic initial value, UCB, and gradient bandits? How would you rank them in terms of regret?

Obviously, there is no single algorithm that is superior to all the other in all cases. In other words, there is no free lunch. However, based on our experiments, we can compare the different algorithms. these 4 algos have a similarity in the sense that they try to explore the different actions. This exploration is done in different way by different algorithm. For example, epslion-greedy (standard version) has a constant epsilon probability to explore the environment, while the optimistic initial value algo only explores in the beginning. the effect of the initial value disappear across time. in this sense, if the environment is non-stationnary, epsilon greedy would be a better alternative. UCB has also a quasi-constant exploration, however, the more the agent acts in the environmnet the more the confidence interval will shrink and the agent will be more certain of what action to take. Gradient bandit uses an explicit soft policy and samples the action based on the probability which is corrected based in the experience.

Based on our experiments we can rank the different algorithm in terms of regret in this way (best to worse): - UCB - greedy with optimistic initial value - gradient bandit - epsilon-greedy

#---

## 8 Exercise #2

[ ]:
```
# https://github.com/zafarali/emdp#grid-world
! git clone https://github.com/zafarali/emdp.git
```

```
Cloning into 'emdp'…
remote: Enumerating objects: 285, done.
remote: Counting objects: 100% (125/125), done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 285 (delta 76), reused 65 (delta 32), pack-reused 160
Receiving objects: 100% (285/285), 74.78 KiB | 1.31 MiB/s, done.
Resolving deltas: 100% (155/155), done.
```

[ ]:
```
%cd /content/emdp
```

```
/content/emdp
```

[ ]:
```
!pip install -e .
```

```
Obtaining file:///content/emdp
Requirement already satisfied: numpy>=1.9.1 in /usr/local/lib/python3.7/dist-
packages (from emdp==0.0.5) (1.19.5)
Installing collected packages: emdp
```

```
  Running setup.py develop for emdp
Successfully installed emdp-0.0.5
```

### 8.0.1 Building the environment

[ ]:
```
import emdp.gridworld as gw
import matplotlib.pyplot as plt
from emdp.gridworld import GridWorldPlotter
from os import system, name

def build_SB_example(seed=1):

    size = 5
    P = gw.build_simple_grid(size=size, p_success=1)
    # modify P to match dynamics from book.

    P[0, :, :] = 0 # first set the probability of all actions from state 1 to␣
 ↪zero
    P[0, :, 0] = 1

    P[24, :, :] = 0
    P[24, :, 24] = 1 # now set the probability of all actions from state 25 to␣
 ↪zero


    R = - np.ones((P.shape[0], P.shape[1])) # initialize a matrix of size␣
 ↪|S|x|A|
    R[0, :] = 0
    R[24, :] = 0

    p0 = np.ones(P.shape[0])/P.shape[0] # uniform starting probability (assumed)
    gamma = 1

    terminal_states = []
    return gw.GridWorldMDP(P, R, gamma, p0, terminal_states, size, seed)
```

### 8.0.2 Creating a class that has policy evaluation, iteration and value iteration as methods

[ ]:
```
from emdp import actions
import copy
import numpy as np

def softmax(arr):
  numerator = np.exp(arr - max(arr))
  return numerator / np.sum (numerator)
```

```
class GridWorld_value:
  def __init__(self, mdp):
    mdp.reset()
    self.mdp = mdp
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space
    self.actions = [actions.LEFT, actions.RIGHT, actions.UP, actions.DOWN]

  def reset(self):
    self.mdp.reset()
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space

  def convert_to_xy(self,state):
    return ( state//self.mdp.size, state%self.mdp.size)

  def back_up(self, state):
    # we can do this because the policy here is deterministic
    v = 0
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v += self.policy[state,i] * (reward + self.old_V[self.convert_to_xy(np.
↪argmax(next_state))] )
    return v

  def value_iteration_back_up(self, state):
    v = []
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v.append( (reward + self.old_V[self.convert_to_xy(np.argmax(next_state))]␣
↪))
    return v


  def policy_evaluation(self, epsilon=1e-9):
    delta = 0
    i = 0
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = self.back_up(state)
```

```
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
      if diff<epsilon:
        break
    print('Policy evaluation done')

  def policy_improvement(self):

    policy =  np.zeros_like(self.policy)
    for state in range(self.mdp.state_space):
      q_value = np.zeros(self.mdp.action_space)

      for i, action in enumerate(self.actions):
        self.mdp.set_current_state_to(self.convert_to_xy(state))
        next_state, reward, done, info = self.mdp.step(action)
        q_value[i] = (reward + self.V[self.convert_to_xy(np.
↪argmax(next_state))] )


      policy[state, np.argmax(q_value)] = 1
    self.policy = policy


  def policy_iteration(self):
    policy_stable = True
    while policy_stable:
      old_policy = copy.deepcopy(self.policy)
      self.policy_evaluation()
      self.policy_improvement()

      policy_stable = np.allclose(old_policy, self.policy, rtol=1e-5)
    self.policy_evaluation()
    print('Policy iteration DONE')

  def value_iteration(self, epsilon=1e-9):
    delta = 0
    i = 0
    self.old_V = np.zeros((mdp.size,mdp.size))
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = np.max(self.
↪value_iteration_back_up(state))
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
```

```
      if diff<epsilon:
        break
    self.policy_improvement()

  def run_episode(self, seed):
    policy = self.policy
    final_state = np.array([0,24])
    curr_state = self.mdp.reset()
    curr_state__ = np.where(curr_state ==1)[0][0]

    cond = curr_state__ in final_state
    cum_reward = []
    i = 0

    while not(cond):
      i += 1

      action = np.argmax(policy[curr_state__])
      old_state = curr_state__

      curr_state, reward, _, _ = self.mdp.step(int(action))

      curr_state__ = np.where(curr_state ==1)[0][0]
      # print('state: {}  | action: {} | next_state: {} | reward: {}'.
↪format(old_state, action, curr_state__, reward))
      cum_reward.append(reward)
      cond = curr_state__ in final_state

    return np.sum(cum_reward), i
```

### 8.0.3 Running Policy evaluation for Q1

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.policy_evaluation()

print(Grid.V)
```

```
Policy evaluation done
[[  0.         -22.99999998 -34.33333331 -39.66666663 -41.66666663]
 [-22.99999998 -30.66666664 -36.3333333  -38.99999997 -39.66666663]
 [-34.33333331 -36.3333333  -37.3333333  -36.3333333  -34.33333331]
 [-39.66666663 -38.99999997 -36.3333333  -30.66666664 -22.99999998]
 [-41.66666663 -39.66666663 -34.33333331 -22.99999998   0.        ]]
```

$$q_\pi(11, \text{down}) = R + \gamma * V_\pi(\text{next state} = 16) = -1 + 1 * (-39) = -40$$

$$q_\pi(7, \text{down}) = R + \gamma * V_\pi(\text{next state} = 12) = -1 + 1 * (-37.33) = -38.33$$

# 9 Question2

Assumption: the policy is deterministic

#Question3

### 9.0.1 Running Policy iteration and displaying the optimal policy and the value function associated to it

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)

Grid.policy_iteration()
print('*'*50)
print('THE OPTIMAL POLICY FOUND IS THE FOLLOWING')
print(Grid.policy)

print('*'*50)
print('THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING')
print(Grid.V)
```

```
Policy evaluation done
Policy evaluation done
Policy iteration DONE
**************************************************
THE OPTIMAL POLICY FOUND IS THE FOLLOWING
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
```

### 2.1 Q2.1 8 / 8

✓ - 0 pts Correct

```
      if diff<epsilon:
        break
    self.policy_improvement()

  def run_episode(self, seed):
    policy = self.policy
    final_state = np.array([0,24])
    curr_state = self.mdp.reset()
    curr_state__ = np.where(curr_state ==1)[0][0]

    cond = curr_state__ in final_state
    cum_reward = []
    i = 0

    while not(cond):
      i += 1

      action = np.argmax(policy[curr_state__])
      old_state = curr_state__

      curr_state, reward, _, _ = self.mdp.step(int(action))

      curr_state__ = np.where(curr_state ==1)[0][0]
      # print('state: {}  | action: {} | next_state: {} | reward: {}'.
↪format(old_state, action, curr_state__, reward))
      cum_reward.append(reward)
      cond = curr_state__ in final_state

    return np.sum(cum_reward), i
```

### 8.0.3 Running Policy evaluation for Q1

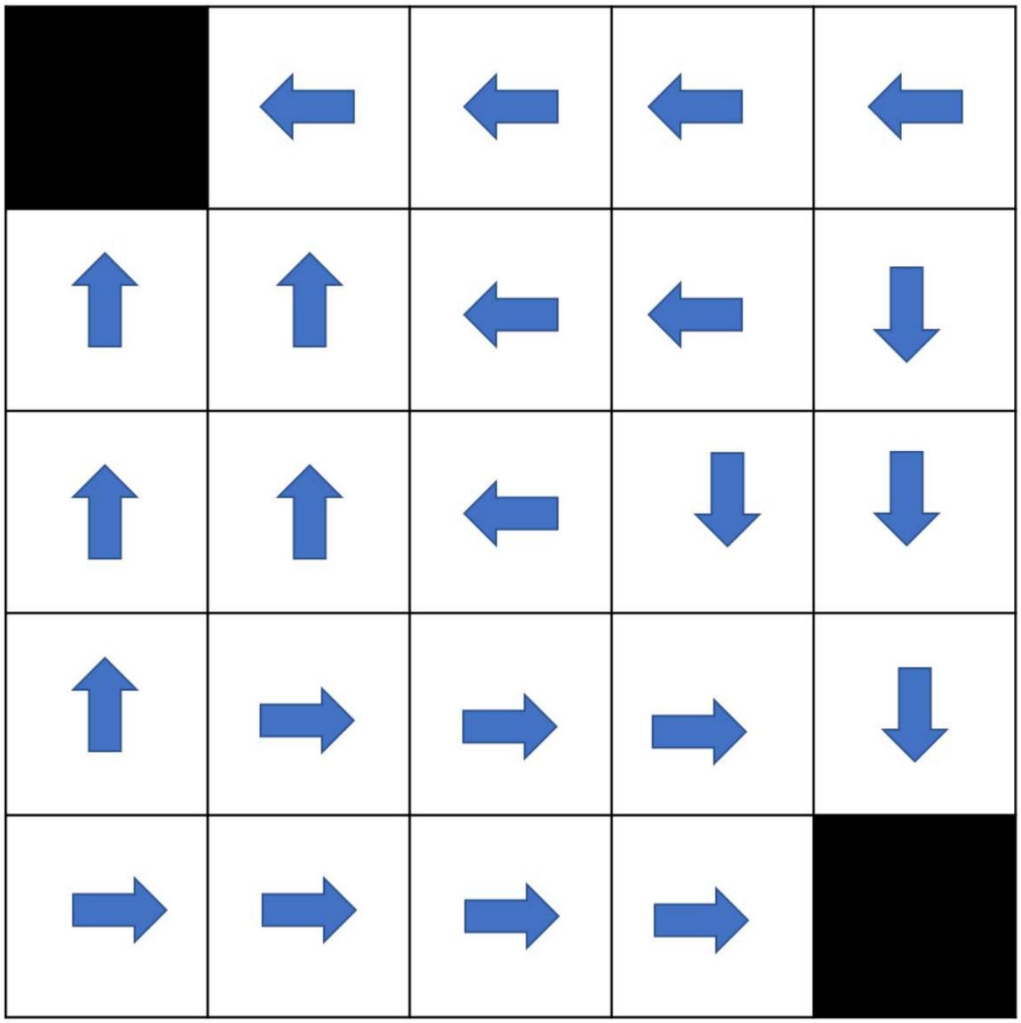
[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.policy_evaluation()

print(Grid.V)
```

```
Policy evaluation done
[[  0.         -22.99999998 -34.33333331 -39.66666663 -41.66666663]
 [-22.99999998 -30.66666664 -36.3333333  -38.99999997 -39.66666663]
 [-34.33333331 -36.3333333  -37.3333333  -36.3333333  -34.33333331]
 [-39.66666663 -38.99999997 -36.3333333  -30.66666664 -22.99999998]
 [-41.66666663 -39.66666663 -34.33333331 -22.99999998   0.        ]]
```

$$q_\pi(11, \text{down}) = R + \gamma * V_\pi(\text{next state} = 16) = -1 + 1 * (-39) = -40$$

## Question2

Assumption: the policy is deterministic



### 2.2 Q2.2 4 / 4

✓ - 0 pts Correct

```
class GridWorld_value:
  def __init__(self, mdp):
    mdp.reset()
    self.mdp = mdp
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space
    self.actions = [actions.LEFT, actions.RIGHT, actions.UP, actions.DOWN]

  def reset(self):
    self.mdp.reset()
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space

  def convert_to_xy(self,state):
    return ( state//self.mdp.size, state%self.mdp.size)

  def back_up(self, state):
    # we can do this because the policy here is deterministic
    v = 0
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v += self.policy[state,i] * (reward + self.old_V[self.convert_to_xy(np.
↪argmax(next_state))] )
    return v

  def value_iteration_back_up(self, state):
    v = []
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v.append( (reward + self.old_V[self.convert_to_xy(np.argmax(next_state))]␣
↪))
    return v


  def policy_evaluation(self, epsilon=1e-9):
    delta = 0
    i = 0
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = self.back_up(state)
```

```
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
      if diff<epsilon:
        break
    print('Policy evaluation done')

  def policy_improvement(self):

    policy =  np.zeros_like(self.policy)
    for state in range(self.mdp.state_space):
      q_value = np.zeros(self.mdp.action_space)

      for i, action in enumerate(self.actions):
        self.mdp.set_current_state_to(self.convert_to_xy(state))
        next_state, reward, done, info = self.mdp.step(action)
        q_value[i] = (reward + self.V[self.convert_to_xy(np.
↪argmax(next_state))] )


      policy[state, np.argmax(q_value)] = 1
    self.policy = policy


  def policy_iteration(self):
    policy_stable = True
    while policy_stable:
      old_policy = copy.deepcopy(self.policy)
      self.policy_evaluation()
      self.policy_improvement()

      policy_stable = np.allclose(old_policy, self.policy, rtol=1e-5)
    self.policy_evaluation()
    print('Policy iteration DONE')

  def value_iteration(self, epsilon=1e-9):
    delta = 0
    i = 0
    self.old_V = np.zeros((mdp.size,mdp.size))
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = np.max(self.
↪value_iteration_back_up(state))
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
```

```
      if diff<epsilon:
        break
    self.policy_improvement()

  def run_episode(self, seed):
    policy = self.policy
    final_state = np.array([0,24])
    curr_state = self.mdp.reset()
    curr_state__ = np.where(curr_state ==1)[0][0]

    cond = curr_state__ in final_state
    cum_reward = []
    i = 0

    while not(cond):
      i += 1

      action = np.argmax(policy[curr_state__])
      old_state = curr_state__

      curr_state, reward, _, _ = self.mdp.step(int(action))

      curr_state__ = np.where(curr_state ==1)[0][0]
      # print('state: {}  | action: {} | next_state: {} | reward: {}'.
↪format(old_state, action, curr_state__, reward))
      cum_reward.append(reward)
      cond = curr_state__ in final_state

    return np.sum(cum_reward), i
```

### 8.0.3 Running Policy evaluation for Q1

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.policy_evaluation()

print(Grid.V)
```

```
Policy evaluation done
[[  0.         -22.99999998 -34.33333331 -39.66666663 -41.66666663]
 [-22.99999998 -30.66666664 -36.3333333  -38.99999997 -39.66666663]
 [-34.33333331 -36.3333333  -37.3333333  -36.3333333  -34.33333331]
 [-39.66666663 -38.99999997 -36.3333333  -30.66666664 -22.99999998]
 [-41.66666663 -39.66666663 -34.33333331 -22.99999998   0.        ]]
```

$$q_\pi(11, \text{down}) = R + \gamma * V_\pi(\text{next state} = 16) = -1 + 1 * (-39) = -40$$

$$q_\pi(7, \text{down}) = R + \gamma * V_\pi(\text{next state} = 12) = -1 + 1 * (-37.33) = -38.33$$

# 9 Question2

Assumption: the policy is deterministic

#Question3

### 9.0.1 Running Policy iteration and displaying the optimal policy and the value function associated to it

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)

Grid.policy_iteration()
print('*'*50)
print('THE OPTIMAL POLICY FOUND IS THE FOLLOWING')
print(Grid.policy)

print('*'*50)
print('THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING')
print(Grid.V)
```

```
Policy evaluation done
Policy evaluation done
Policy iteration DONE
**************************************************
THE OPTIMAL POLICY FOUND IS THE FOLLOWING
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
```

```
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]
**************************************************
THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING
[[ 0. -1. -2. -3. -4.]
 [-1. -2. -3. -4. -3.]
 [-2. -3. -4. -3. -2.]
 [-3. -4. -3. -2. -1.]
 [-4. -3. -2. -1.  0.]]
```

### 9.0.2 Running 5 episodes for 5 seeds

[ ]:
```python
for i in range(5):
  seed = np.random.randint(np.random.randint(1000000))
  np.random.seed(seed)
  mdp = build_SB_example(seed)

  Grid = GridWorld_value(mdp)
  Grid.policy_iteration()

  print('*' *30)
  print('The seed used is: {}'.format(seed))
  cum_reward_avg = []
  iter_avg = []
  for j in range(5):
    cum_reward, iter = Grid.run_episode(seed)
    cum_reward_avg.append(cum_reward)
    iter_avg.append(iter)
    # print('The cumulative reward is: {}'.format(cum_reward))
    # print('The number of iteration is: {}'.format(iter))
  print('THE MEAN OVER THE CUM REWARD IS: {}'.format(np.mean(cum_reward_avg)))
  print('THE MEAN OVER THE NBR OF ITERATIONS IS: {}'.format(np.mean(iter_avg)))
```

```
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 56247
THE MEAN OVER THE CUM REWARD IS: -1.8
THE MEAN OVER THE NBR OF ITERATIONS IS: 1.8
Policy evaluation done
```

```
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 226963
THE MEAN OVER THE CUM REWARD IS: -2.8
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.8
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 2456
THE MEAN OVER THE CUM REWARD IS: -1.4
THE MEAN OVER THE NBR OF ITERATIONS IS: 1.4
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 38446
THE MEAN OVER THE CUM REWARD IS: -2.4
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.4
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 65008
THE MEAN OVER THE CUM REWARD IS: -1.8
THE MEAN OVER THE NBR OF ITERATIONS IS: 1.8
```

### 9.0.3 Running Value iteration and displaying the optimal policy and the value function associated to it

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.value_iteration()
print('*'*50)
print('THE OPTIMAL POLICY FOUND IS THE FOLLOWING')
print(Grid.policy)

print('*'*50)
print('THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING')
print(Grid.V)
```

```
**************************************************
THE OPTIMAL POLICY FOUND IS THE FOLLOWING
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
```

### 2.3 Q2.3(a) 12 / 12

✓ - 0 pts Correct

```
class GridWorld_value:
  def __init__(self, mdp):
    mdp.reset()
    self.mdp = mdp
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space
    self.actions = [actions.LEFT, actions.RIGHT, actions.UP, actions.DOWN]

  def reset(self):
    self.mdp.reset()
    self.V = np.zeros((mdp.size,mdp.size))
    self.old_V = np.zeros((mdp.size,mdp.size))
    self.policy = np.ones((mdp.state_space, mdp.action_space)) /mdp.action_space

  def convert_to_xy(self,state):
    return ( state//self.mdp.size, state%self.mdp.size)

  def back_up(self, state):
    # we can do this because the policy here is deterministic
    v = 0
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v += self.policy[state,i] * (reward + self.old_V[self.convert_to_xy(np.
↪argmax(next_state))] )
    return v

  def value_iteration_back_up(self, state):
    v = []
    for i, action in enumerate(self.actions):
      self.mdp.set_current_state_to(self.convert_to_xy(state))
      next_state, reward, done, info = self.mdp.step(action)
      v.append( (reward + self.old_V[self.convert_to_xy(np.argmax(next_state))]␣
↪))
    return v


  def policy_evaluation(self, epsilon=1e-9):
    delta = 0
    i = 0
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = self.back_up(state)
```

```
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
      if diff<epsilon:
        break
    print('Policy evaluation done')

  def policy_improvement(self):

    policy =  np.zeros_like(self.policy)
    for state in range(self.mdp.state_space):
      q_value = np.zeros(self.mdp.action_space)

      for i, action in enumerate(self.actions):
        self.mdp.set_current_state_to(self.convert_to_xy(state))
        next_state, reward, done, info = self.mdp.step(action)
        q_value[i] = (reward + self.V[self.convert_to_xy(np.
↪argmax(next_state))] )


      policy[state, np.argmax(q_value)] = 1
    self.policy = policy


  def policy_iteration(self):
    policy_stable = True
    while policy_stable:
      old_policy = copy.deepcopy(self.policy)
      self.policy_evaluation()
      self.policy_improvement()

      policy_stable = np.allclose(old_policy, self.policy, rtol=1e-5)
    self.policy_evaluation()
    print('Policy iteration DONE')

  def value_iteration(self, epsilon=1e-9):
    delta = 0
    i = 0
    self.old_V = np.zeros((mdp.size,mdp.size))
    while True:
      i += 1
      for state in range(self.mdp.state_space):
        self.V[self.convert_to_xy(state)] = np.max(self.
↪value_iteration_back_up(state))
      diff = np.max(np.abs(self.old_V - self.V))

      self.old_V = copy.deepcopy(self.V)
```

```
      if diff<epsilon:
        break
    self.policy_improvement()

  def run_episode(self, seed):
    policy = self.policy
    final_state = np.array([0,24])
    curr_state = self.mdp.reset()
    curr_state__ = np.where(curr_state ==1)[0][0]

    cond = curr_state__ in final_state
    cum_reward = []
    i = 0

    while not(cond):
      i += 1

      action = np.argmax(policy[curr_state__])
      old_state = curr_state__

      curr_state, reward, _, _ = self.mdp.step(int(action))

      curr_state__ = np.where(curr_state ==1)[0][0]
      # print('state: {}  | action: {} | next_state: {} | reward: {}'.
↪format(old_state, action, curr_state__, reward))
      cum_reward.append(reward)
      cond = curr_state__ in final_state

    return np.sum(cum_reward), i
```

### 8.0.3 Running Policy evaluation for Q1

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.policy_evaluation()

print(Grid.V)
```

```
Policy evaluation done
[[  0.         -22.99999998 -34.33333331 -39.66666663 -41.66666663]
 [-22.99999998 -30.66666664 -36.3333333  -38.99999997 -39.66666663]
 [-34.33333331 -36.3333333  -37.3333333  -36.3333333  -34.33333331]
 [-39.66666663 -38.99999997 -36.3333333  -30.66666664 -22.99999998]
 [-41.66666663 -39.66666663 -34.33333331 -22.99999998   0.        ]]
```

$$q_\pi(11, \text{down}) = R + \gamma * V_\pi(\text{next state} = 16) = -1 + 1 * (-39) = -40$$

```
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 226963
THE MEAN OVER THE CUM REWARD IS: -2.8
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.8
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 2456
THE MEAN OVER THE CUM REWARD IS: -1.4
THE MEAN OVER THE NBR OF ITERATIONS IS: 1.4
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 38446
THE MEAN OVER THE CUM REWARD IS: -2.4
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.4
Policy evaluation done
Policy evaluation done
Policy iteration DONE
******************************
The seed used is: 65008
THE MEAN OVER THE CUM REWARD IS: -1.8
THE MEAN OVER THE NBR OF ITERATIONS IS: 1.8
```

### 9.0.3 Running Value iteration and displaying the optimal policy and the value function associated to it

[ ]:
```
mdp = build_SB_example()

Grid = GridWorld_value(mdp)
Grid.value_iteration()
print('*'*50)
print('THE OPTIMAL POLICY FOUND IS THE FOLLOWING')
print(Grid.policy)

print('*'*50)
print('THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING')
print(Grid.V)
```

```
**************************************************
THE OPTIMAL POLICY FOUND IS THE FOLLOWING
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
```

```
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [0. 1. 0. 0.]
 [1. 0. 0. 0.]]
**************************************************
THE OPTIMAL VALUE FUNCTION FOUND IS THE FOLLOWING
[[ 0. -1. -2. -3. -4.]
 [-1. -2. -3. -4. -3.]
 [-2. -3. -4. -3. -2.]
 [-3. -4. -3. -2. -1.]
 [-4. -3. -2. -1.  0.]]
```

#### 9.0.4 Running 5 episodes for 5 seeds

[ ]:
```python
for i in range(5):
  seed = np.random.randint(np.random.randint(1000000))
  np.random.seed(seed)
  mdp = build_SB_example(seed)

  Grid = GridWorld_value(mdp)
  Grid.value_iteration()
  print('*' *30)
  print('The seed used is: {}'.format(seed))
  cum_reward_avg = []
  iter_avg = []
  for j in range(5):
    cum_reward, iter = Grid.run_episode(seed)
    cum_reward_avg.append(cum_reward)
    iter_avg.append(iter)
```

```
    # print('The cumulative reward is: {}'.format(cum_reward))
    # print('The number of iteration is: {}'.format(iter))
  print('THE MEAN OVER THE CUM REWARD IS: {}'.format(np.mean(cum_reward_avg)))
  print('THE MEAN OVER THE NBR OF ITERATIONS IS: {}'.format(np.mean(iter_avg)))
```

```
******************************
The seed used is: 104019
THE MEAN OVER THE CUM REWARD IS: -3.0
THE MEAN OVER THE NBR OF ITERATIONS IS: 3.0
******************************
The seed used is: 156016
THE MEAN OVER THE CUM REWARD IS: -3.2
THE MEAN OVER THE NBR OF ITERATIONS IS: 3.2
******************************
The seed used is: 73658
THE MEAN OVER THE CUM REWARD IS: -2.0
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.0
******************************
The seed used is: 213407
THE MEAN OVER THE CUM REWARD IS: -2.6
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.6
******************************
The seed used is: 590509
THE MEAN OVER THE CUM REWARD IS: -2.4
THE MEAN OVER THE NBR OF ITERATIONS IS: 2.4
```

## 10 Conclusion

The policy obtained using policy iteration, value iteration and the one I was expecting are quasi-similar. In fact, there are some states where 2 or more actions had all the same q_value but the agent chose one randomly, thus we obtained slight differences.

[ ]:

### 2.4 Q2.3(b) 6 / 6

✓ - 0 pts Correct