Mount the Google Drive onto the Colab as the storage location.

Following the instructions returned from the below cell. You will click a web link and select the google account you want to mount, then copy the authorication code to the blank, press enter.

```
# This must be run within a Google Colab environment
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Append the directory location where you upload the start code folder (In this problem, *RLalgs*) to the sys.path

E.g. dir = '/content/drive/My Drive/RL/.', start code folder is inside "RL" folder.

```
import sys
sys.path.append('/content/gdrive/MyDrive/2021fall/e6885/coding_assignment/.')
# sys.path.append('</dir/to/start/code/folder/.>')
```

Your code should remain in the block marked by
############################
YOUR CODE STARTS HERE
YOUR CODE ENDS HERE
############################
Please don't edit anything outside the block.

```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
import numpy as np
import random
import matplotlib.pyplot as plt
import gym
```

1. Incremental Implementation of Average

We've finished the incremental implementation of average for you. Please call the function estimate with 1/step step size and fixed step size to compare the difference between this two on a simulated Bandit problem.

```
from RLalgs.utils import estimate
random.seed(6885)
numTimeStep = 10000
q_h = np.zeros(numTimeStep + 1) # Q Value estimate with 1/step step size
q_f = np.zeros(numTimeStep + 1) # Q value estimate with fixed step size
FixedStepSize = 0.5 #A large number to exaggerate the difference
for step in range(1, numTimeStep + 1):
        if step < numTimeStep / 2:
            # gauss distribution
                r = random.gauss(mu = 1, sigma = 0.1)
        else:
                r = random.gauss(mu = 3, sigma = 0.1)

        #TIPS: Call function estimate defined in ./RLalgs/utils.py
        ############################
        # YOUR CODE STARTS HERE
        q_h[step] = estimate(q_h[step-1], 1/step, r)
        q_f[step] = estimate(q_h[step-1], FixedStepSize, r)
        # YOUR CODE ENDS HERE
        ############################

q_h = q_h[1:]
q_f = q_f[1:]
```

    RLalgs is a package containing Reinforcement Learning algorithms Epsilon-Greedy, Policy Iter

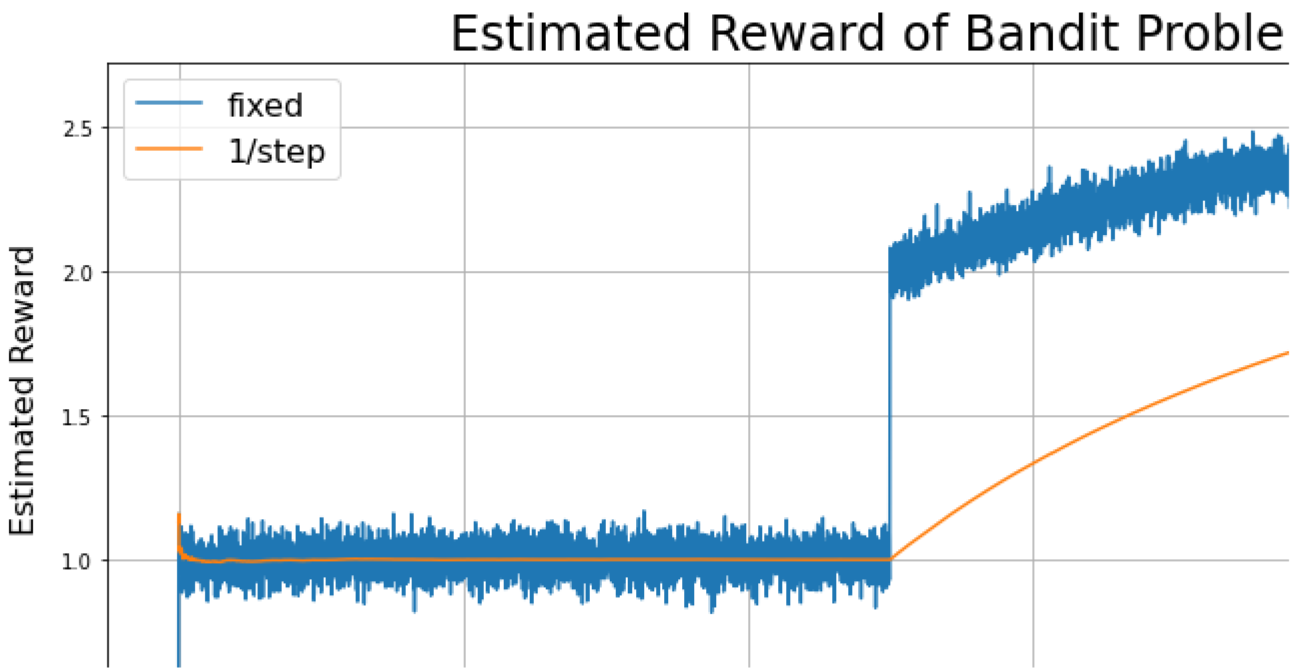
Plot the two Q value estimates. (Please include a title, labels on both axes, and legends)

```
############################
# YOUR CODE STARTS HERE

plt.figure(figsize=(14,6))

# fixed stpsize
plt.plot(range(numTimeStep), q_f, label='fixed')
# 1/step stepsize
plt.plot(range(numTimeStep), q_h, label='1/step')
plt.grid()
plt.legend(fontsize=16)
plt.title('Estimated Reward of Bandit Problem', fontsize=24)
plt.ylabel('Estimated Reward', fontsize=16, labelpad=10)
plt.xlabel('Steps', fontsize=16, labelpad=10);

# YOUR CODE ENDS HERE
############################
```



2. ϵ-Greedy for Exploration

In Reinforcement Learning, we are always faced with the dilemma of exploration and exploitation. ϵ-Greedy is a trade-off between them. You are gonna implement Greedy and ϵ-Greedy. We combine these two policies in one function by treating Greedy as ϵ-Greedy where $\epsilon = 0$. Edit the function epsilon_greedy in ./RLalgs/utils.py.

```
from RLalgs.utils import epsilon_greedy
np.random.seed(6885) #Set the seed to cancel the randomness
q = np.random.normal(0, 1, size = 5)
###############################
# YOUR CODE STARTS HERE
greedy_action = epsilon_greedy(q, 0) #Use epsilon = 0 for Greedy
e_greedy_action = epsilon_greedy(q, 0.1) #Use epsilon = 0.1
# YOUR CODE ENDS HERE
###############################
print('Values:')
print(q)
print('Greedy Choice =', greedy_action)
print('Epsilon-Greedy Choice =', e_greedy_action)
```

```
Values:
[ 0.61264537  0.27923079 -0.84600857  0.05469574 -1.09990968]
Greedy Choice = 0
Epsilon-Greedy Choice = 0
```

You should get the following results.

Values:

[0.61264537 0.27923079 -0.84600857 0.05469574 -1.09990968]

Greedy Choice = 0

3. Frozen Lake Environment

```
env = gym.make('FrozenLake-v0')
```

3.1 Derive Q value from V value

Edit function action_evaluation in ./RLalgs/utils.py.
TIPS: $q(s,a) = \sum_{s',r} p(s',r|s,a)(r + \gamma v(s'))$

```
from RLalgs.utils import action_evaluation
v = np.ones(16)
q = action_evaluation(env = env.env, gamma = 1, v = v)
print('Action values:')
print(q)
```

```
Action values:
[[1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.         1.         1.        ]
 [1.         1.33333333 1.33333333 1.33333333]
 [1.         1.         1.         1.        ]]
```

You should get Q values all equal to one except at State 14

Pseudo-codes of the following four algorithms can be found on Page 80, 83, 130, 131 of the Sutton's book.

3.2 Model-based RL algorithms

```
from RLalgs.utils import action_evaluation, action_selection, render
```

3.2.1 Policy Iteration

Edit the function policy_iteration and relevant functions in ./RLalgs/pi.py to implement the

```
from  RLalgs.pi  import  policy_iteration
V,  policy,  numIterations  =  policy_iteration(env  =  env.env,  gamma  =  1,  max_iteration  =
print('State  values:')
print(V)
print('Number  of  iterations  to  converge  =',  numIterations)
```

```
State values:
[0.82352769 0.82352712 0.82352673 0.82352653 0.82352786 0.
 0.52941059 0.         0.82352813 0.82352848 0.76470507 0.
 0.         0.88235229 0.94117614 0.        ]
Number of iterations to converge = 500
```

You should get values close to:
State values:
[0.82352774 0.8235272 0.82352682 0.82352662 0.82352791 0.
0.52941063 0. 0.82352817 0.82352851 0.76470509 0.
0. 0.88235232 0.94117615 0.]

```
#Uncomment  and  run  the  following  to  evaluate  your  result,  comment  them  when  you  ger
#  Q  =  action_evaluation(env  =  env.env,  gamma  =  1,  v  =  V)
#  policy_estimate  =  action_selection(Q)
#  render(env,  policy_estimate)
```

3.2.2 Value Iteration

Edit the function value_iteration and relevant functions in ./RLalgs/vi.py to implement the Value Iteration Algorithm.

```
from  RLalgs.vi  import  value_iteration
V,  policy,  numIterations  =  value_iteration(env  =  env.env,  gamma  =  1,  max_iteration  =
print('State  values:')
print(V)
print('Number  of  iterations  to  converge  =',  numIterations)
```

```
State values:
[0.82352513 0.82352369 0.82352267 0.82352214 0.82352544 0.
 0.5294087  0.         0.82352604 0.82352689 0.76470365 0.
 0.         0.88235115 0.94117554 0.        ]
Number of iterations to converge = 500
```

You should get values close to:
State values:
[0.82352773 0.82352718 0.8235268 0.8235266 0.8235279 0.
0.52941062 0. 0.82352816 0.8235285 0.76470509 0.
0. 0.88235231 0.94117615 0.]

```
#Uncomment and run the following to evaluate your result, comment them when you ger
# Q = action_evaluation(env = env.env, gamma = 1, v = V)
# policy_estimate = action_selection(Q)
# render(env, policy_estimate)
```

3.3 Model free RL algorithms

3.3.1 Q-Learning

Edit the function QLearning in ./RLalgs/ql.py to implement the Q-Learning Algorithm.

```
from RLalgs.ql import QLearning
Q = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e = 0.1)
print('Action values:')
print(Q)
```

```
Action values:
[[2.11199210e-03 3.67909924e-02 3.60097180e-03 1.08141550e-02]
 [4.47010439e-03 2.63382467e-02 3.84696543e-02 1.33505818e-02]
 [8.13408307e-02 2.51285268e-02 3.30812594e-02 1.18327649e-02]
 [2.51288321e-02 3.10232495e-03 7.19206088e-05 4.24684003e-05]
 [6.55202187e-02 4.36773528e-03 8.78594226e-03 5.15825836e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [8.56434774e-02 2.97735169e-02 4.43788385e-02 2.83638073e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [8.00560910e-03 9.28663313e-02 3.80966177e-02 3.16427834e-02]
 [0.00000000e+00 2.39119991e-01 1.11204636e-01 7.51017004e-02]
 [1.09846695e-01 8.27005453e-02 2.97020742e-01 8.02628561e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [6.73159352e-02 9.08349828e-02 7.21099186e-02 2.14149757e-01]
 [1.78006566e-01 6.54237672e-01 3.21100169e-01 2.51990747e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Generally, you should get non-zero action values on non-terminal states.

```
#Uncomment the following to evaluate your result, comment them when you generate the
# env = gym.make('FrozenLake-v1')
# policy_estimate = action_selection(Q)
# render(env, policy_estimate)
```

3.3.2 SARSA

Edit the function SARSA in ./RLalgs/sarsa.py to implement the SARSA Algorithm.

```
from RLalgs.sarsa import SARSA
Q = SARSA(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e = 0.1)
print('Action values:')
print(Q)
```

```
Action values:
[[7.46111235e-02 6.73376815e-02 1.08750153e-01 5.56249585e-02]
 [3.07835188e-02 2.18086590e-02 2.41144029e-02 1.13791892e-01]
 [1.48316881e-01 2.25421919e-02 6.39042944e-02 3.25353974e-02]
 [2.07363577e-02 9.38229063e-03 1.03752717e-02 7.68844233e-02]
 [1.38554506e-01 9.82300893e-02 3.70805732e-02 5.50829298e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.48074228e-01 2.01559919e-02 5.49124151e-02 1.81843732e-04]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [8.18453622e-02 1.21912079e-01 5.07193900e-02 1.81406611e-01]
 [5.73904157e-02 2.33346024e-01 1.31252032e-01 4.75127127e-02]
 [3.14899445e-01 2.88634098e-01 2.13909429e-01 5.06246406e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.17805420e-01 2.08277949e-01 1.68114919e-01 1.52920155e-01]
 [2.21025847e-01 5.08879901e-01 3.64096881e-01 3.97002465e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Generally, you should get non-zero action values on non-terminal states.

```
#Uncomment the following to evaluate your result, comment them when you generate the
# env = gym.make('FrozenLake-v0')
# policy_estimate = action_selection(Q)
# render(env, policy_estimate)
```

3.4 Human

You can play this game if you are interested. See if you can get the frisbee either with or without the model.

```
from RLalgs.utils import human_play
# Uncomment and run the following to play the game, comment it when you generate
# env = gym.make('FrozenLake-v1')
# human_play(env)
```

4. Exploration VS. Exploitation

Try to reproduce Figure 2.2 (the upper one is enough) of the Sutton's book based on the experiment described in Chapter 2.3.

```
from RLalgs.utils import *

class Bandit():
    def __init__(self, num_arm, num_iter, eps):
        """
        Bandit problem following e-greedy policy, the arm machien follow norma

        Inputs:
        - num_arm: the number of arm machine
```

```
        - num_iter: the number of total actions to be taken
        - eps: epsilon in e-greedy policy

        """
        # number of arms
        self.num_arm = num_arm
        # number of total actions to be taken
        self.num_iter = num_iter
        # epsilon
        self.eps = eps
        # record average reward for the whole game
        self.reward = 0
        # self.reward_hist = [0]
        self.reward_hist = np.zeros(num_iter+1)
        self.reward_hist[0] = 0
        # record average reward for each arms
        self.q = np.zeros(num_arm)
        # number of actions taken for the whole game
        self.num_step = 0
        # number of actions taken on each arms
        self.num_step_arm = np.zeros(num_arm)
        # initialize arms
        self.arms = None
        self.init_arms()

    def reset(self, eps, num_iter):
        """
        reset eps and reward
        """
        self.reward = 0
        # self.reward_hist = [0]
        self.reward_hist = np.zeros(num_iter+1)
        self.reward_hist[0] = 0
        self.q = np.zeros_like(self.q)
        self.num_step = 0
        self.num_step_arm = np.zeros_like(self.num_step_arm)
        self.eps = eps
        self.num_iter = num_iter

    def init_arms(self):
        """
        Initialize arms by randomly choose mu and std for normal distribution

        mu follows normal distribution
        std = 1

        store in array self.arms, which has size 2*num_arm
        first row of arms is mu
        second row of arms is std
        """
        assert self.arms == None, 'Arms already defined'

        # first row is mu, secound row is std
        # arms = np.random.rand(2, self.num_arm)
        arms = np.random.normal(0,1,(2,self.num_arm))
```

```

        # arms[0,:] = arms[0,:]*4 - 2
        # arms[1,:] = arms[1,:]*10 + 20
        arms[1,:] = 1

        self.arms = arms

    def take_action(self, a):
        """
        Take action a, range of a is [0,num_arm-1], return the reward of tha

        Input:
        - a: the action to be taken

        Output:
        - r: reward of that action
        """

        return np.random.normal(self.arms[0,a], self.arms[1,a])

    def step(self):
        """
        one step forward, choose the action following the e-greedy policy
        """

        # e-greedy policy
        a = epsilon_greedy(self.q, self.eps)
        # get reward of that action
        r = self.take_action(a)
        # update steps taken
        self.num_step += 1
        self.num_step_arm[a] += 1
        # update the total average reward and the average reward of that ar
        self.reward = estimate(self.reward, 1/self.num_step, r)
        # self.reward_hist.append(self.reward)
        self.q[a] = estimate(self.q[a], 1/self.num_step_arm[a], r)

    def run(self):
        """
        play the game for iteration times

        Output:
        - history of total average reward
        """
        for i in range(self.num_iter):
            self.step()
            self.reward_hist[i+1] = self.reward

        return self.reward_hist



```

You should get curves similar to that in the book.

```
#  Plot  the  average  reward
############################


num_arms  =  10
num_iter  =  1000
num_repeat  =  1000

hist_0  =  np.zeros(num_iter+1)
hist_001  =  np.zeros(num_iter+1)
hist_01  =  np.zeros(num_iter+1)


for  i  in  range(1,  num_repeat+1):
        eps  =  0
        bandit  =  Bandit(num_arms,  num_iter,  0)
        hist_0  +=  bandit.run()

        #  eps  =  0.01
        bandit.reset(0.01,  num_iter)
        hist_001  +=  bandit.run()


        #  eps  =  0.1
        bandit.reset(0.1,  num_iter)
        hist_01  +=  bandit.run()

        if  i%100  ==  0:
                print('Repeated  %d  times'  %  (i))

hist_0  /=  num_repeat
hist_001  /=  num_repeat
hist_01  /=  num_repeat

plt.figure()
plt.plot(hist_0,  'g',  label="$\epsilon=0$  (Greedy)")
plt.plot(hist_001,  'r',  label="$\epsilon=0.01$")
plt.plot(hist_01,  'b',  label="$\epsilon=0.1$")
plt.title("Average  $\epsilon-greedy$  rewards  after  %d  Repeats"  %  (num_repeat))
plt.xlabel("Steps")
plt.ylabel("Average  Reward")
plt.legend();


#  YOUR  CODE  ENDS  HERE
############################
```

```
Repeated 100 times
Repeated 200 times
Repeated 300 times
Repeated 400 times
Repeated 500 times
Repeated 600 times
Repeated 700 times
Repeated 800 times
Repeated 900 times
Repeated 1000 times
```

✓ 0s completed at 11:36 PM