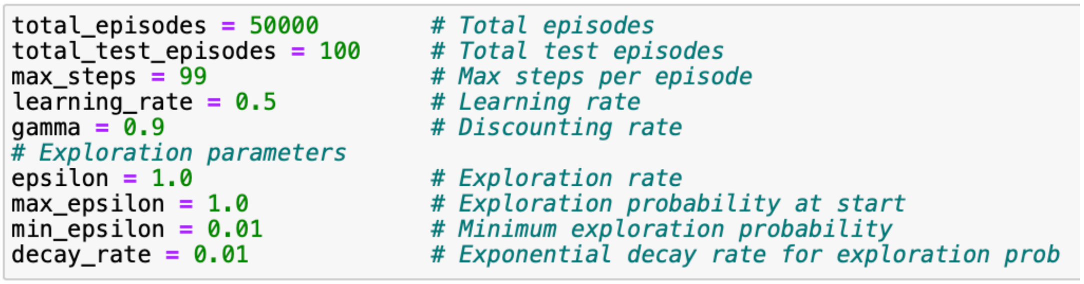
Deep Q-Learning with an Atari-like game - Breakout-v0

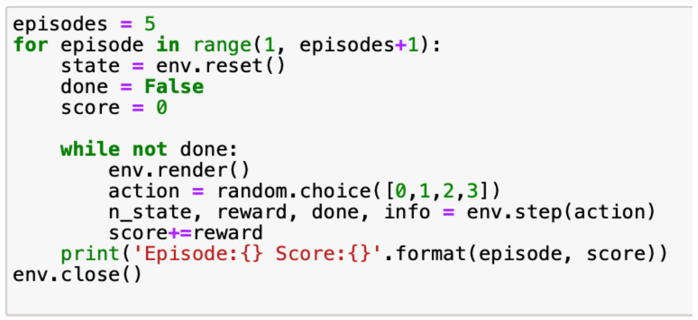
https://gym.openai.com/envs/Breakout-v0/

1. Establish a baseline performance. How well did your Deep Q-learning do on your problem? (5 Points)

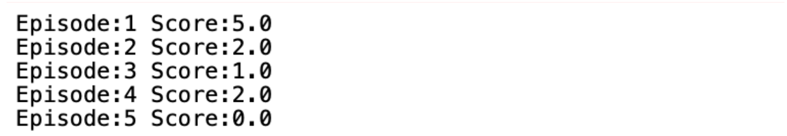
With this baseline performance, our RL program with the Breakout-v0 gives us a mean score of 2.0. Simply throughout the 5 episodes run, the rolling average of 5 consecutive episodes was only 2.0 which is considered somewhat bad.



```
total_episodes = 50000          # Total episodes
total_test_episodes = 100       # Total test episodes
max_steps = 99                  # Max steps per episode
learning_rate = 0.5             # Learning rate
gamma = 0.9                     # Discounting rate
# Exploration parameters
epsilon = 1.0                   # Exploration rate
max_epsilon = 1.0               # Exploration probability at start
min_epsilon = 0.01              # Minimum exploration probability
decay_rate = 0.01               # Exponential decay rate for exploration prob
```



```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = random.choice([0,1,2,3])
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
```



```
Episode:1 Score:5.0
Episode:2 Score:2.0
Episode:3 Score:1.0
Episode:4 Score:2.0
Episode:5 Score:0.0
```

2. What are the states, the actions, and the size of the Q-table? (5 Points)

The states include:

game_state: Is the game ongoing / lost / won or just started

ball: The x and y coordinates of the ball

ball_vel: The x and y components of the ball’s velocity

paddle: The x position of the paddle

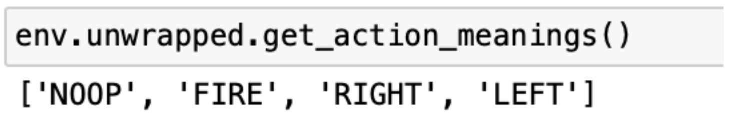
bricks: Array of the coordinates of the remaining bricks

time: The number of frames since the game started

score: The current score of the game

lives: The number of lives left%

The actions = ['NOOP', 'FIRE', 'RIGHT', 'LEFT']



```
env.unwrapped.get_action_meanings()
```

```
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

where each element separately in the list mean

0: do nothing

1: fire ball to start game

2: move right

3: move left

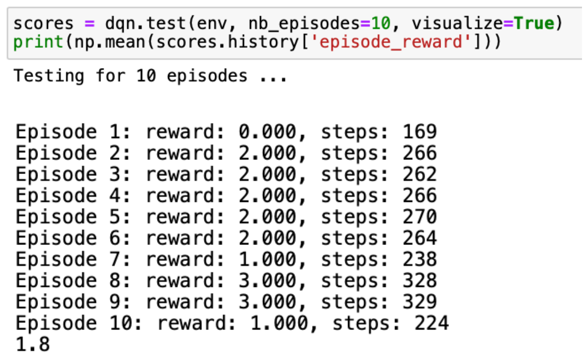
The size of the Q-table:

Q-table		Actions			
		NOOP	FIRE	RIGHT	LEFT
States	0				
	. .				
	. .				
	210				

(Total 210 states)

3. What are the rewards? Why did you choose them? (5 Points)

We went through 10000 steps in 39 episodes. We can see its mean scores of episode reward when we tested it for 10 episodes. It is helpful for us to review how performance it is without having to browse so many episodes. But in our case, we trained 10000 steps which were not enough to show its optimal point. We can only see its changes in this running range.

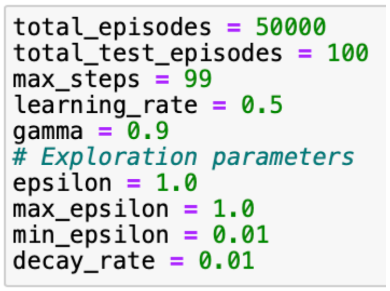


```
scores = dqn.test(env, nb_episodes=10, visualize=True)
print(np.mean(scores.history['episode_reward']))
```

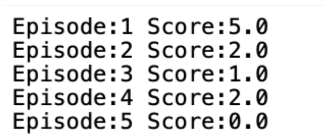
```
Testing for 10 episodes ...

Episode 1: reward: 0.000, steps: 169
Episode 2: reward: 2.000, steps: 266
Episode 3: reward: 2.000, steps: 262
Episode 4: reward: 2.000, steps: 266
Episode 5: reward: 2.000, steps: 270
Episode 6: reward: 2.000, steps: 264
Episode 7: reward: 1.000, steps: 238
Episode 8: reward: 3.000, steps: 328
Episode 9: reward: 3.000, steps: 329
Episode 10: reward: 1.000, steps: 224
1.8
```

4. How did you choose alpha and gamma in the Bellman equation? Try at least one additional value for alpha and gamma. How did it change the baseline performance? (5 Points)

Alpha also known as the learning rate. α should be in the range of 0 to 1, which determines to what extent newly acquired information overrides old information. The higher the learning rate, it quickly replaces the new q value. When I chose the hyperparameters of the model, I used the middle value to operate it because we did not how much do it influence our model. Gamma also known as the discounting rate. γ is discounting for past rewards, which determines the importance of future rewards. which is set between 0 and 1. I just set α = 0.5 and γ= 0.9 at the first.

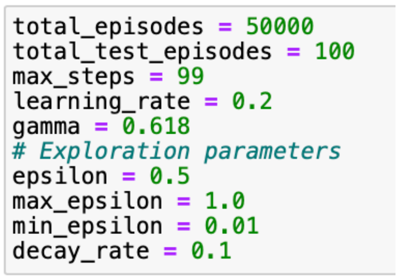


```
total_episodes = 50000
total_test_episodes = 100
max_steps = 99
learning_rate = 0.5
gamma = 0.9
# Exploration parameters
epsilon = 1.0
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.01
```

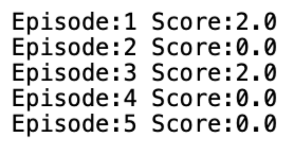


```
Episode:1 Score:5.0
Episode:2 Score:2.0
Episode:3 Score:1.0
Episode:4 Score:2.0
Episode:5 Score:0.0
```

Then I tried an additional value of alpha and gamma, the baseline performance has deteriorated. After I adjusted these two hyperparameters, we can find that its consecutive average score decreased from 2.0 to 0.8.



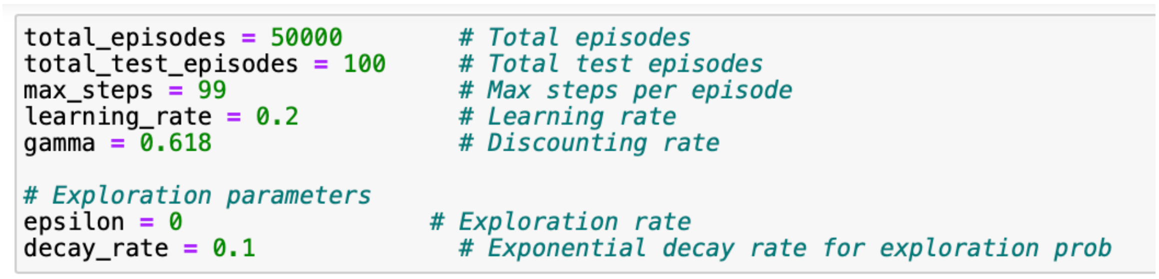
```
total_episodes = 50000
total_test_episodes = 100
max_steps = 99
learning_rate = 0.2
gamma = 0.618
# Exploration parameters
epsilon = 0.5
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.1
```



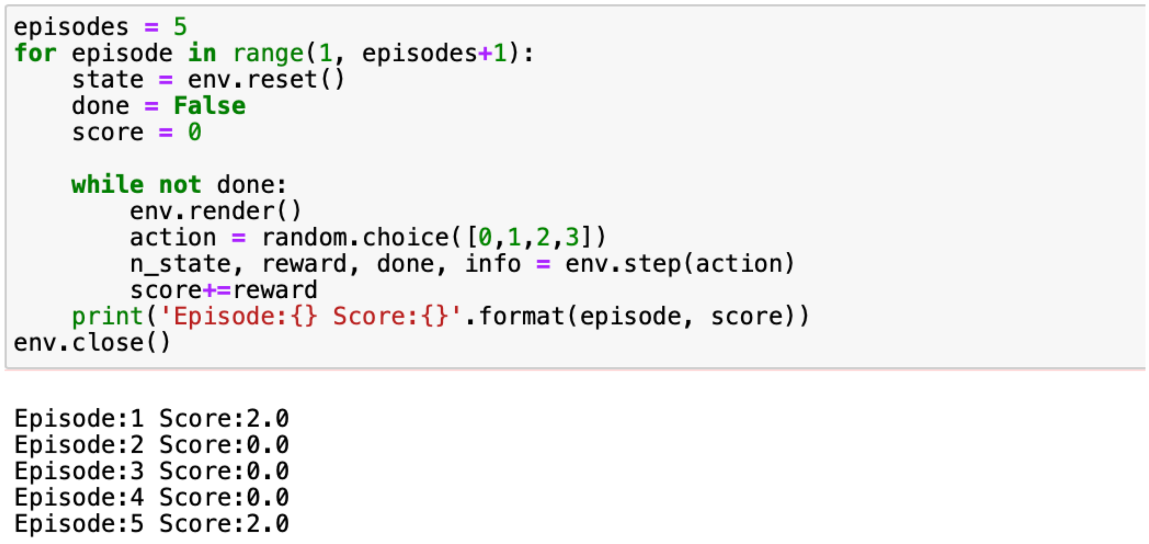
```
Episode:1 Score:2.0
Episode:2 Score:0.0
Episode:3 Score:2.0
Episode:4 Score:0.0
Episode:5 Score:0.0
```

5. Try a policy other than ε-greedy. How did it change the baseline performance? (5 Points)

I tried reinforcement learning with the conventional greedy policy instead of the epsilon greedy approach. For the conventional greedy policy, the parameter epsilon should be 0 which means there is no exploration and always be greedy. After running episodes simply, we can see that the average of 5 consecutive episodes is only 0.8 which is much lower than the ε-greedy model. As we know, the greedy method converged to a sub-optimal point with very little exploration, while the epsilon-greedy model continues to explore and converge to the optimal action. The performance of the conventional greedy approach might keep the certain reward because it will converge to its optimal point.



```
total_episodes = 50000        # Total episodes
total_test_episodes = 100     # Total test episodes
max_steps = 99                # Max steps per episode
learning_rate = 0.2           # Learning rate
gamma = 0.618                 # Discounting rate

# Exploration parameters
epsilon = 0                 # Exploration rate
decay_rate = 0.1              # Exponential decay rate for exploration prob
```

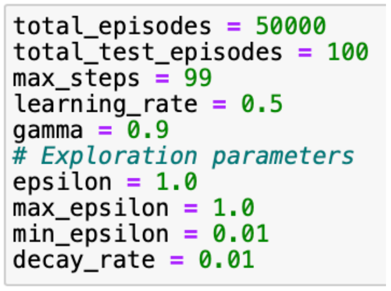


```
episodes = 5
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = random.choice([0,1,2,3])
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
env.close()
```

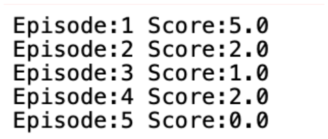
```
Episode:1 Score:2.0
Episode:2 Score:0.0
Episode:3 Score:0.0
Episode:4 Score:0.0
Episode:5 Score:2.0
```

6. How did you choose your decay rate and starting epsilon? Try at least one additional value for epsilon and the decay rate. How did it change the baseline performance? What is the value of epsilon when if you reach the max steps per episode? (5 Points)

By choosing these parameters, I introduce a random value for epsilon. For example, if epsilon = 0.5 then we are selecting random actions with 0.5 probability regardless of the actual q value. The decay rate is how much you are updating the Q value with each step. I set it to a low value because I want it fully to exploit and stabilize the policy and setting it too high will prevent it from learning.

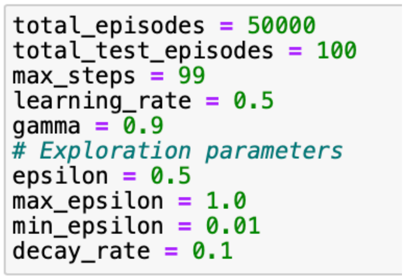


```
total_episodes = 50000
total_test_episodes = 100
max_steps = 99
learning_rate = 0.5
gamma = 0.9
# Exploration parameters
epsilon = 1.0
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.01
```

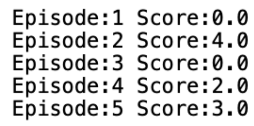


```
Episode:1 Score:5.0
Episode:2 Score:2.0
Episode:3 Score:1.0
Episode:4 Score:2.0
Episode:5 Score:0.0
```

Then I tried an additional value of epsilon and the decay rate, the baseline performance has indeed changed. After I adjusted these two variables, we can find that its score was better than before. Its consecutive average score improved to 4.5, so these two parameters might increase the score and make exploration better. We can know that the explored parameters have successfully helped our model here.



```
total_episodes = 50000
total_test_episodes = 100
max_steps = 99
learning_rate = 0.5
gamma = 0.9
# Exploration parameters
epsilon = 0.5
max_epsilon = 1.0
min_epsilon = 0.01
decay_rate = 0.1
```



```
Episode:1 Score:0.0
Episode:2 Score:4.0
Episode:3 Score:0.0
Episode:4 Score:2.0
Episode:5 Score:3.0
```

7. What is the average number of steps taken per episode? (5 Points)

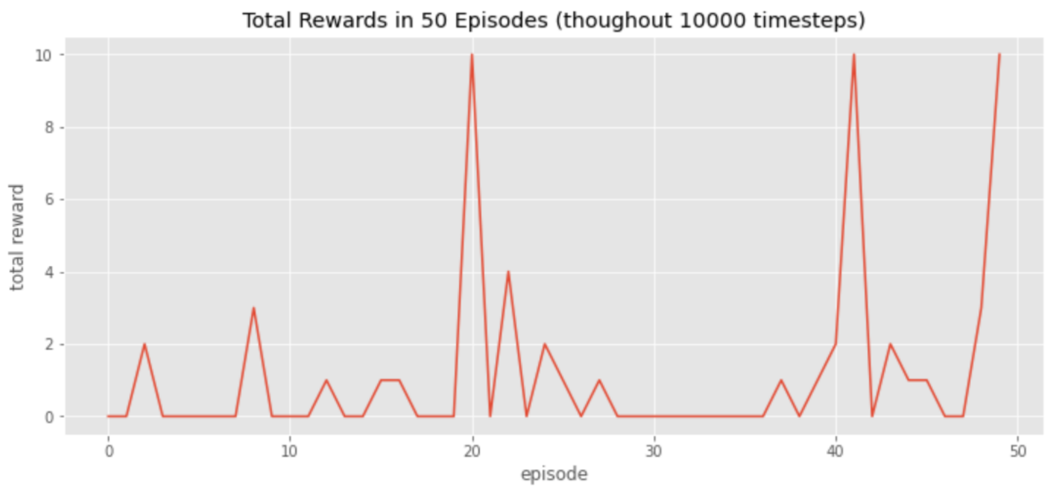
In this Q-learning model, we trained for 10000 steps, and it split those steps into 39 episodes. Therefore, on average, it takes 256.41 steps per episode.

```
   354/10000: episode: 1, duration: 9.048s, episode steps: 354, steps per second:  39, episode reward:  3.000,
 mean reward:  0.008 [ 0.000,  1.000], mean action: 1.599 [0.000, 3.000],  loss: --, mean_q: --, mean_eps: --
   623/10000: episode: 2, duration: 6.849s, episode steps: 269, steps per second:  39, episode reward:  2.000,
 mean reward:  0.007 [ 0.000,  1.000], mean action: 1.379 [0.000, 3.000],  loss: --, mean_q: --, mean_eps: --
```

…… …… ……

```
 9522/10000: episode: 38, duration: 279.507s, episode steps: 232, steps per second:   1, episode reward:  1.0
00, mean reward:  0.004 [ 0.000,  1.000], mean action: 1.539 [0.000, 3.000],  loss: 1.727466, mean_q: 30.5474
11, mean_eps: 0.153505
 9747/10000: episode: 39, duration: 268.767s, episode steps: 225, steps per second:   1, episode reward:  0.0
00, mean reward:  0.000 [ 0.000,  0.000], mean action: 1.711 [0.000, 3.000],  loss: 1.428768, mean_q: 30.7479
22, mean_eps: 0.132940
done. took 10329.462 seconds
```

After training through 10000 timestep, we have not been able to get a very good result because we have not trained more episodes. We need more training to get better and continuous improvement results. Make a plot of points Timestep, Reward.



8. Does Q-learning use value-based or policy-based iteration? (5 Points)
Explain, not a yes or no question.

Yes, Q-learning use value-based iteration.
Q-learning is a value-based but off-policy, and we can modify the value iteration algorithm to solve for the Q-values. And it provides a means of finding the optimal policy.

9. Could you use SARSA for this problem? (5 Points)
Explain, not a yes or no question.

Yes, we could use SARSA for this Atari Breakout problem. We can use an ε-greedy policy for π: with probability ε the action is chosen randomly, and the action is chosen to satisfy a = arg max$_a$ Qπ(s, a). On each (s, a, r, s', a') tuple, Qπ(s, a) moves towards an estimate of the true value of being in state s and taking action a. This value is the sum of discounted rewards the agent.

10. What is meant by the expected lifetime value in the Bellman equation? (5 Points)
Explain, not a yes or no question.

In the Bellman equation, the expected lifetime value is the value for being in a certain state that the agent will receive from the current time step t to the end of the task.

11. When would SARSA likely do better than Q-learning? (5 Points)
Explain, not a yes or no question.

SARSA is on policy while Q Learning is off policy.
In SARSA, we take actual action and in Q Learning we take the action with highest reward. When training neural networks, Q-learning has a higher per-sample variance than SARSA and may suffer from problems converging as a result. If the agent learns online, and we care about rewards gained whilst learning, then SARSA may be a better choice.
SARSA is typically preferable in situations where we care about the agent's performance during the process of learning or generating experience, while Q-learning would be preferable in situations where we do not care about the agent's performance during the training process. In practice the last point can make a big difference if mistakes are costly - e.g., you are training a robot not in a simulation, but in the real world. You may prefer a more conservative learning algorithm that avoids high risk if there was real time and money at stake if the robot was damaged.

12. How does SARSA differ from Q-learning? (5 Points)
Details including pseudocode and math.

The biggest difference between Q-learning and SARSA is that Q-learning is off-policy, and SARSA is on-policy. The most important difference between the two is how Q is updated after each action. SARSA uses the Q' following a ε-greedy policy exactly, as A' is drawn from it. In contrast, Q-learning uses the maximum Q' over all possible actions for the next step. The rules are as follows:
SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Q Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

where st, at and rt are state, action and reward at time step t and γ is a discount factor.

Q-learning chooses next action A' and updating Q by different policies. It evaluates π while following another policy μ, so it's an off-policy algorithm. In contrast, SARSA uses π all the time, hence it is an on-policy algorithm.

13. Explain the Q-learning algorithm. (5 Points)
Details including pseudocode and math.

Q-learning is an off-policy reinforcement learning algorithm that seeks to find the best action to take given the current state. The Q-learning function learns from actions that are out of the current policy, like taking random actions, so a policy isn't needed. It seeks to learn a policy that maximizes the total reward.

Pseudocode for Q-learning:

```
*Input:* Given starting state distribution $D_0$;
*Initialization:* Take $\hat{}Q$ to be an empty array
**repeat** for e=1, 2, … **until** change in $\hat{}Q$ is small
    $s_0 \sim D_0$.
    Select step sizes $\alpha_0, \alpha_1, \ldots$
    **repeat** for t=1, 2, … **until** epoch is terminated
        Select an action $a_t$.
        Observe the next state $s_{t+1}$ and the reward $R(s_t,a_t,s_{t+1})$.
        Take $\delta_t: =(R(s_t,a_t,s_{t+1})+\gamma \max_{a'\in A} \hat{}Q(s_{t+1},a'))-\hat{}Q(s_t,a_t)$
        $Q(s_t,a_t) \leftarrow \hat{}Q(s_t,a_t)+\alpha_t\delta_t$
```

Math for Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

14. Explain the SARSA algorithm. (5 Points)
Details including pseudocode and math.

SARSA is an on-policy algorithm where S an action, A is taken and the agent gets a reward, R and ends up in next state, S1 and takes action, A1 in S1. It chooses an action following the same current policy and updates its Q-values

Pseudocode for Q-learning:

```
Initialize all Q(s, a) arbitrarily
```

```
For all episodes
    Initialize s
Choose a using policy derived from Q, e.g., ε-greedy
    Repeat
        Take action a, observe r and s'
        Choose a' using policy derived from Q, e.g., ε-greedy
        Update Q(s, a):
            $Q(s, a) \leftarrow Q(s, a) + \eta(r + \gamma Q(s', a') - Q(s, a))$
         $s \leftarrow s', a \leftarrow a'$
    Until s is terminal state
```

Math for SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

15. What code is yours and what have you adapted? (5 Points)
You must explain what code you wrote and what you have done that is different. Failure to cite ANY code will result in a zero for this section.

For details, see the section of References at the bottom of the notebook

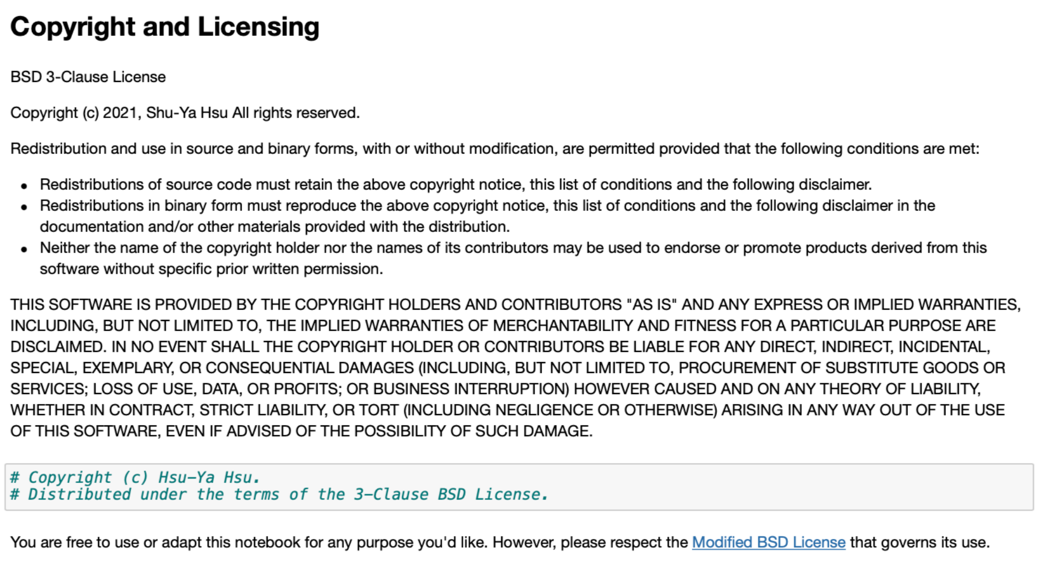
GitHub for my notebook: https://github.com/leahsu/CSYE7370

16. Did I explain my code clearly? (10 Points)

For details, see the explanations in the notebook.

17. Did I explain my licensing clearly? (5 Points)

The license is at the bottom of the notebook. The following screenshot for reference.



Copyright and Licensing

```
# Copyright (c) Hsu-Ya Hsu.
# Distributed under the terms of the 3-Clause BSD License.
```

18. Professionalism (10 Points)

Variable naming, style guide, conduct, behavior, and attitude.