CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep dari pembelajaran mesin kemudian di terapkan pada permasalahan – (C2)
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin untuk mendapatkan pemecahan maslaah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 12

REINFORCEMENT LEARNING

DESKRIPSI TEMA

Mahasiswa mempelajari pembelajaran tidak terbimbing dengan algoritma pembelajaran trial dan error yaitu Q-Learning dan SARSA

CAPAIAN PEMBELAJARAN MINGGUAN (SUB CPMK)

Mahasiswa dapat menerapkan algoritma Q Learning dan SARSA sehingga bisa menghasilkan model terbaik

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. So, the main points in reinforcement learning are :

- Input : the input should be an initial state from which the model will start
- Output : there are many possible output as there are variety of solution to a particular problem
- Training : the training is based upon the input. The model will return a state and the user will decide to reward or punish the model based on its output
- The mode keeps continues to learn
- The best solution is decided based on the maximum reward

Q LEARNING ALGORITHM

Q-learning is a basic from a Reinforcement Learning which uses Q-values (also called action values) to iteratively improve the behavior of learning agent.

a. **Q-Values or Action Values** : Q-Values are defined for states and actions. Q(S,A) is an estimation of how good is it to take the action A at the state S. This estimation of Q(S,A) will be iteratively computed using the TD-Update rule which we will sedd in the upcoming sections
b. **Rewards and Episodes :** an agent over the course of its lifetime starts from a start state, makes a number of transition from its current state to a next state based on its choice of action and also the environment the agent is interacting in. At every

step of transition, the agent from a state takes an action, observe a reward from the environment and then transits to another state. If at any point of time the agent ends up in one of the terminating states that means there are no further transition possible. This is said to be the completion of an episode.

c. **Temporal Difference or TD-Update** : the TD-Update rule can be represent as follows :

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',S') - Q(S,A))$$

This update rule to estimate the value of Q is applied at every time step of the agents interactions with the environment. The terms used are explained below :

- **S** : current state of the agent
- **A** : current action picked according to some policy
- **S'** : next state where the agent ends up
- **A'** : next best action to be picked using current Q-value estimation, i.e : pick the action with the maximum Q-value in the next state
- **R** : current reward observed from the environment in Response of current action
- $\boldsymbol{\gamma(> 0\ and \leq 1)}$: discounting factor for the future rewards. Future rewards are less valuable than current rewards so they must be discounted. Since Q-value is an estimation of expected rewards from a state, discounting rule applies here as well
- $\boldsymbol{\alpha}$: step length taken to update the estimation of Q(S,A)

d. **Choosing the action to take using ε-greedy policy** : ε-greedy policy is a very simple policy of choosing action using the current Q-value estimation. It goes as follows :

- With probability (1-ε) choose the action which has the highest Q-value
- With probability (ε) choose any action at random

Now, with all the explanation, we do some work with Q-Learning Algorithm. We will use OpenAI gym environment to train our Q-Learning model.

1. Installing OpenAI gym

```
pip install gym
```

There will be 2 helper files which need to be inserting in the working directory. There are plotting and environment we created, named windy_gridworld (***inserted in your elearning directory – week 11***)

2. Import required libraries

```python
import gym
import itertools
import matplotlib
import matplotlib.style
import numpy as np
import pandas as pd
import sys
```

3. Import the environment :

```python
from collections import defaultdict
from windy_gridworld import WindyGridworldEnv
import plotting

matplotlib.style.use('ggplot')
```

4. Create gym environment

```python
env = WindyGridworldEnv()
```

5. Make the ε-greedy policy

```python
def createEpsilonGreedyPolicy(Q, epsilon, num_actions):
    def policyFunction(state):

        Action_probabilities = np.ones(num_actions,
                dtype = float) * epsilon / num_actions

        best_action = np.argmax(Q[state])
        Action_probabilities[best_action] += (1.0 - epsilon)
        return Action_probabilities

    return policyFunction

```

   We create an epsilon – greedy policy based on given Q-function and epsilon. Returns a function that takes the state as an input and return the probabilities for each action in the form of numpy array of length of the action space (set of possible actions).

6. Build Q-learning model

```python
def qLearning(env, num_episodes, discount_factor = 1.0,
                            alpha = 0.6, epsilon = 0.1):
    # Action value function
    # A nested dictionary that maps
    # state -> (action -> action-value).
    Q = defaultdict(lambda: np.zeros(env.action_space.n))

    # Keeps track of useful statistics
    stats = plotting.EpisodeStats(
        episode_lengths = np.zeros(num_episodes),
        episode_rewards = np.zeros(num_episodes))

    # Create an epsilon greedy policy function
    # appropriately for environment action space
    policy = createEpsilonGreedyPolicy(Q, epsilon, env.action_space.n)

    # For every episode
    for ith_episode in range(num_episodes):

        # Reset the environment and pick the first action
        state = env.reset()

        for t in itertools.count():

            # get probabilities of all actions from current state
            action_probabilities = policy(state)

            # choose action according to
            # the probability distribution
            action = np.random.choice(np.arange(
                      len(action_probabilities)),
                       p = action_probabilities)

            # take action and get reward, transit to next state
            next_state, reward, done, _ = env.step(action)

            # Update statistics
            stats.episode_rewards[ith_episode] += reward
            stats.episode_lengths[ith_episode] = t

            # TD Update
            best_next_action = np.argmax(Q[next_state])
            td_target = reward + discount_factor * Q[next_state][best_next_action]
            td_delta = td_target - Q[state][action]
            Q[state][action] += alpha * td_delta

            # done is True if episode terminated
            if done:
                break

            state = next_state

    return Q, stats
```

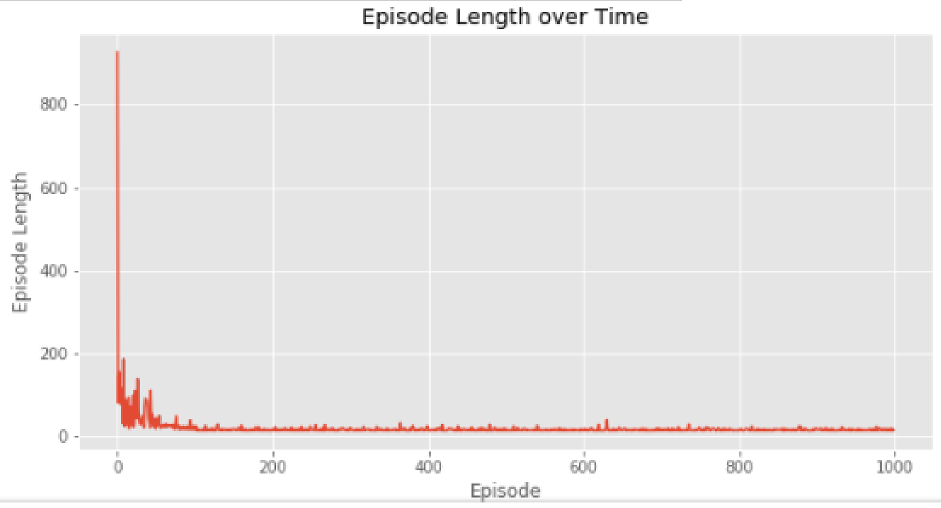
Q-learning algorithm : off – policy TD control. Finds the optimal greedy policy while improving following an epsilon – greedy policy.

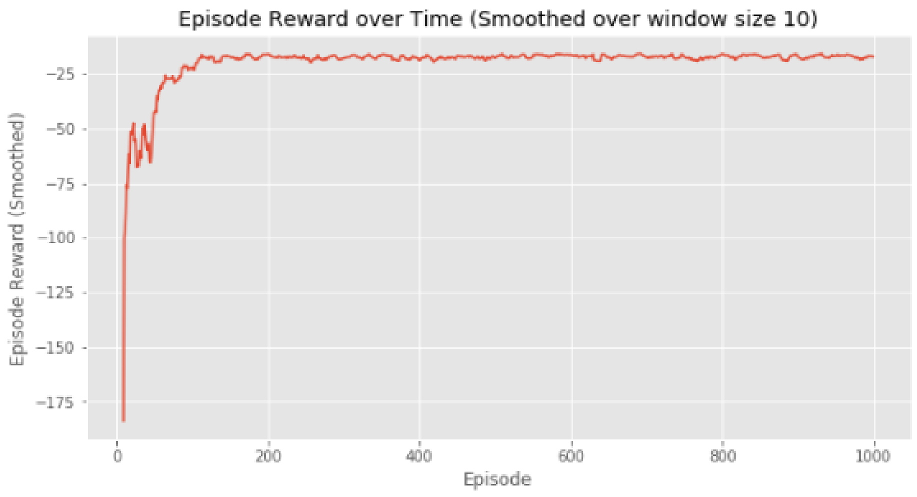
7. Train the model

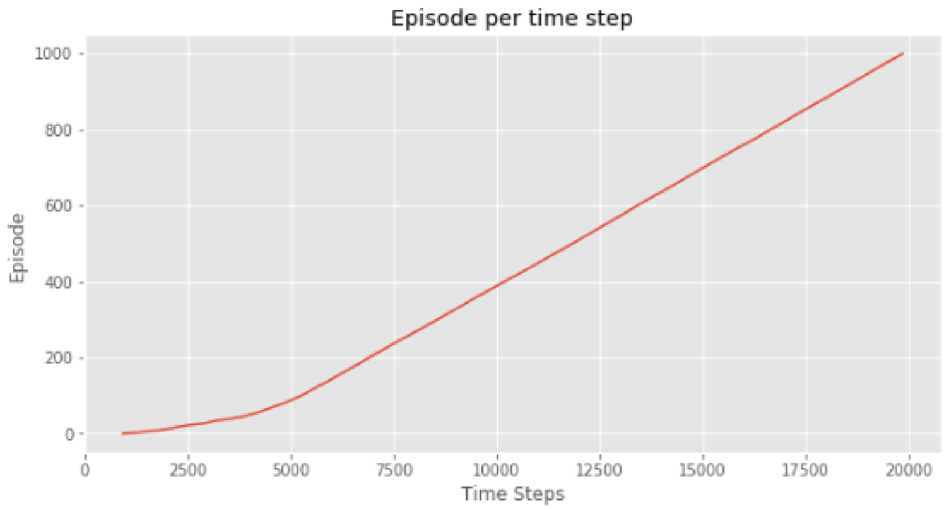
```
Q, stats = qLearning(env, 1000)
```

8. Plot important statistics

```
plotting.plot_episode_stats(stats)
```







We can see that in Episode Reward over time plot that episode rewards progressively increase over time and ultimately levels out at a high reward per episode value which indicates that the agent has learn to maximize its total reward earned in an episode by behaving optimally at every state.

After we know about basic of Reinforcement Learning, let's we see the demonstration with basic implemented.

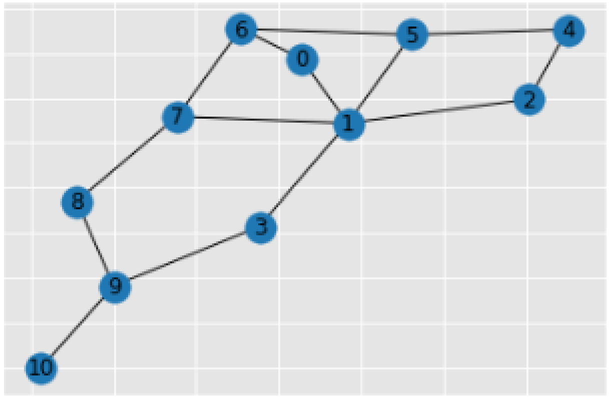
9. Importing the required libraries

```python
import numpy as np
import pylab as pl
import networkx as nx
```

~~10. Defining and visualizing the graph~~

```
edges = [(0, 1), (1, 5), (5, 6), (5, 4), (1, 2),
         (1, 3), (9, 10), (2, 4), (0, 6), (6, 7),
         (8, 9), (7, 8), (1, 7), (3, 9)]
```

```
goal =10
G = nx.Graph()
G.add_edges_from(edges)
pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G,pos)
nx.draw_networkx_edges(G,pos)
nx.draw_networkx_labels(G,pos)
pl.show()
```



The above graph may not look the same on reproduction of the code because the networkx library in python produces a random graph from the given code.

11. Defining the reward the system for the bot

```
MATRIX_SIZE = 11
M = np.matrix(np.ones(shape =(MATRIX_SIZE, MATRIX_SIZE)))
M *= -1

for point in edges:
    print(point)
    if point[1] == goal:
        M[point] = 100
    else:
        M[point] = 0

    if point[0] == goal:
        M[point[::-1]] = 100
    else:
        M[point[::-1]]= 0
        # reverse of point

M[goal, goal]= 100
print(M)
```

```
[[ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.   0.   0.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.]
 [ -1.  -1.   0.  -1.  -1.   0.  -1.  -1.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.  -1.]
 [  0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.  -1.]
 [ -1.   0.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.   0.  -1.   0.  -1.]
 [ -1.  -1.  -1.   0.  -1.  -1.  -1.  -1.   0.  -1. 100.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.   0. 100.]]
```

12. Defining some utility functions to be used in the training

```
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))

gamma = 0.75
# learning parameter
initial_state = 1
```

Determines the available action for a given state

```
def available_actions(state):
    current_state_row = M[state, ]
    available_action = np.where(current_state_row >= 0)[1]
    return available_action

available_action = available_actions(initial_state)
```

Choose one of the available actions at random

```
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action


action = sample_next_action(available_action)
```

Updating

```
def update(current_state, action, gamma):

  max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  if (np.max(Q) > 0):
    return(np.sum(Q / np.max(Q)*100))
  else:
    return (0)
```

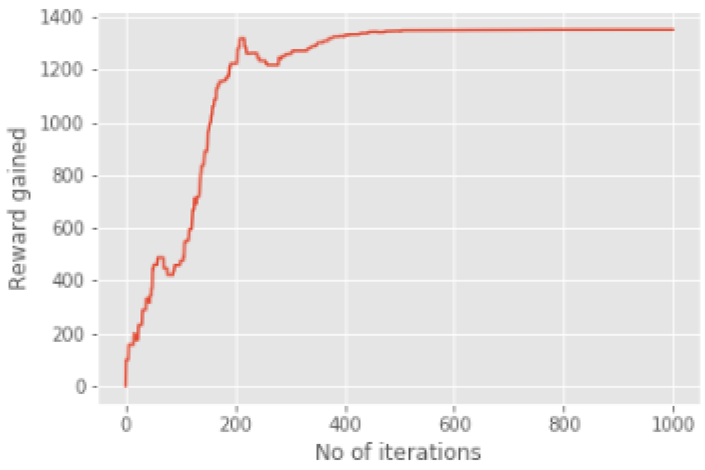
```python
update(initial_state, action, gamma)
```

13. Training and evaluating the bot using the Q-Matrix

```python
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
    scores.append(score)
```

```python
# Testing
current_state = 0
steps = [current_state]

while current_state != 10:

    next_step_index = np.where(Q[current_state, ] == np.max(Q[current_state, ]))[1]
    if next_step_index.shape[0] > 1:
        next_step_index = int(np.random.choice(next_step_index, size = 1))
    else:
        next_step_index = int(next_step_index)
    steps.append(next_step_index)
    current_state = next_step_index

print("Most efficient path:")
print(steps)

pl.plot(scores)
pl.xlabel('No of iterations')
pl.ylabel('Reward gained')
pl.show()
```

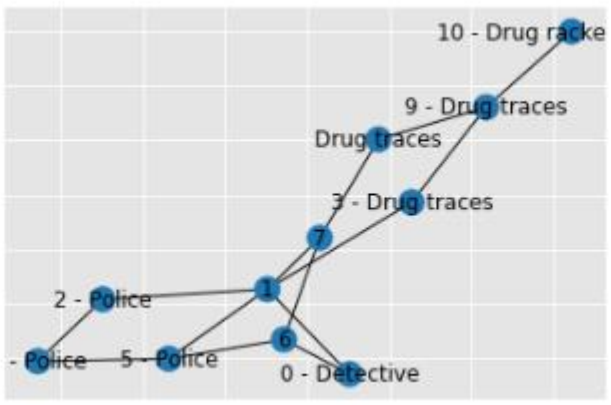
```
Most efficient path:
[0, 1, 3, 9, 10]
```



Now, let’s bring this bot more realistic setting. Let us imagine that the bot is a detective and is trying to find out the location of a large drug racket. He or she naturally concludes that the drug seller will not sell their products in a location which is known to be frequented by the police and the selling location are near the location of the drug racket. Also, the sellers leave a trace of their products where they sell it and this can help the detective in finding out the required location. We want to train our bot to find the location using these Environment Clues.

14. Defining and visualizing the new graph with the environment clues

```
police = [2, 4, 5]
drug_traces = [3, 8, 9]

G = nx.Graph()
G.add_edges_from(edges)
mapping = {0:'0 - Detective', 1:'1', 2:'2 - Police', 3:'3 - Drug traces',
           4:'4 - Police', 5:'5 - Police', 6:'6', 7:'7', 8:'Drug traces',
           9:'9 - Drug traces', 10:'10 - Drug racket location'}

H = nx.relabel_nodes(G, mapping)
pos = nx.spring_layout(H)
nx.draw_networkx_nodes(H, pos, node_size =[200, 200, 200, 200, 200, 200, 200, 200])
nx.draw_networkx_edges(H, pos)
nx.draw_networkx_labels(H, pos)
pl.show()
```



15. Defining some utility functions for the training process

```
Q = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_police = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
env_drugs = np.matrix(np.zeros([MATRIX_SIZE, MATRIX_SIZE]))
initial_state = 1
```

```python
def available_actions(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]
    return av_action
```

```python
def sample_next_action(available_actions_range):
    next_action = int(np.random.choice(available_action, 1))
    return next_action

```

```python
def collect_environmental_data(action):
    found = []
    if action in police:
        found.append('p')
    if action in drug_traces:
        found.append('d')
    return (found)


available_action = available_actions(initial_state)
action = sample_next_action(available_action)
```

```python
def update(current_state, action, gamma):
  max_index = np.where(Q[action, ] == np.max(Q[action, ]))[1]
  if max_index.shape[0] > 1:
      max_index = int(np.random.choice(max_index, size = 1))
  else:
      max_index = int(max_index)
  max_value = Q[action, max_index]
  Q[current_state, action] = M[current_state, action] + gamma * max_value
  environment = collect_environmental_data(action)
  if 'p' in environment:
    env_police[current_state, action] += 1
  if 'd' in environment:
    env_drugs[current_state, action] += 1
  if (np.max(Q) > 0):
    return(np.sum(Q / np.max(Q)*100))
  else:
    return (0)
update(initial_state, action, gamma)
```

```python
def available_actions_with_env_help(state):
    current_state_row = M[state, ]
    av_action = np.where(current_state_row >= 0)[1]

    # if there are multiple routes, dis-favor anything negative
    env_pos_row = env_matrix_snap[state, av_action]

    if (np.sum(env_pos_row < 0)):
        temp_av_action = av_action[np.array(env_pos_row)[0]>= 0]
        if len(temp_av_action) > 0:
            av_action = temp_av_action
    return av_action
```

16. Visualizing the environmental metrics

```python
scores = []
for i in range(1000):
    current_state = np.random.randint(0, int(Q.shape[0]))
    available_action = available_actions(current_state)
    action = sample_next_action(available_action)
    score = update(current_state, action, gamma)
```

```python
print('Police Found')
print(env_police)
print('')
print('Drug traces Found')
print(env_drugs)
```

```
Police Found
[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 17.  0.  0. 21.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 55.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0. 48.  0.  0. 44.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0. 34.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0. 17.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]


 Drug traces Found
 [[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0. 21.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0. 41.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0. 33.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0. 47.  0.]
  [ 0.  0.  0. 29.  0.  0.  0.  0. 22.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0. 54.  0.]]
```

Q-Learning Example : Self – Driving Cab

The smart cab job is to pick up the passenger at one location and drop them off in another. Here are a few things that we'd love our SmartCab to take care of :

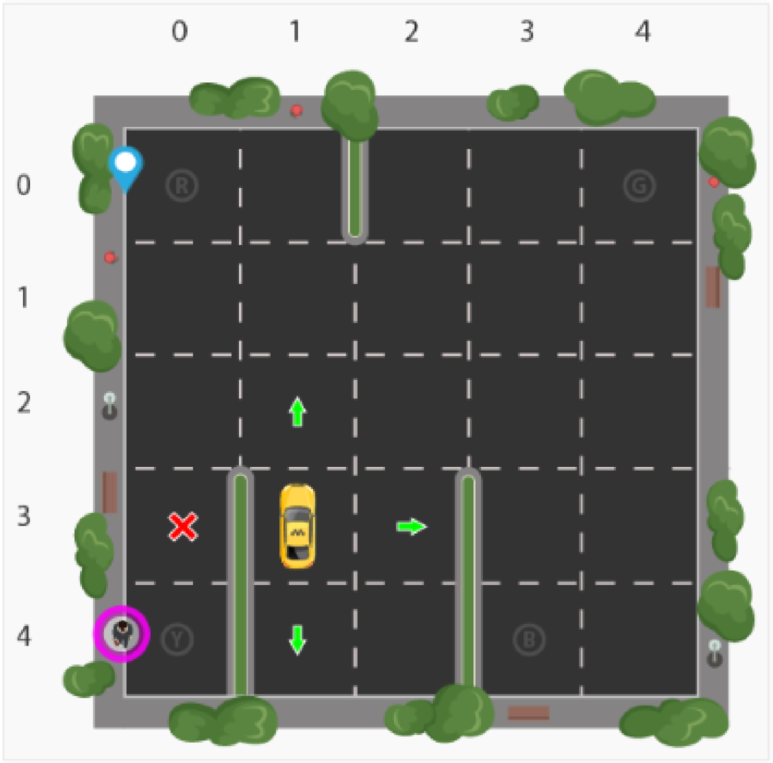
- Drop off the passenger to the right location
- Save passengers time by taking minimum time possible to drop off
- Take care of passenger safety and traffic rules

Rewards :

- The agent should receive the high positive reward for a successful drop off because this behavior is highly desired
- The agent should be penalized if it tries to drop off passenger in wrong locations
- The agents should get a slight negative reward for not making it to the destination after every time-step. "Slight" negative because we would prefer our agent to reach late instead of making wrong moves trying to reach to the destination as fast as possible

State space :

Let's say we have a training area for our SmartCab where we are teaching it to transport people in a parking lot to four different locations (R, G, Y, B) :



Let’s assume SmartCab is the only vehicle in this parking lot. We can break up the parking lot into 5x5 grid which gives us possible taxi locations. These 25 locations are one part of our state space. Notice the current location state of taxi is coordinate (3,1).

We’ll also notice there are (4) location that we can pick up and drop off a passenger : R G Y B ([0,0], [0,4], [4,0],[4,3]) in (row,col) coordinates. Our illustrated passenger is in location Y and they wish to go to location R.

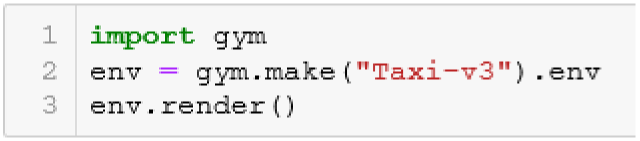
When we also account for (1) additional passenger state of being inside the taxi, we can take all combinations of passenger locations and destination locations to come to a total number of states for our taxi environment; there’s (4) destinations and 5 (4+1) passenger locations. So, our taxi environment has 5x5x5x4 = 500 total possible states.

The agent encounters one of the 500 states and it takes an action. The action in our case can be to move in a direction or decide to pickup/drop off a passenger.

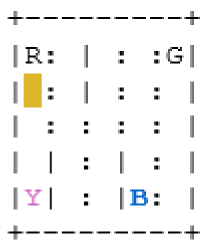
In other words, we have six possible actions : south, north, east. West, pickup, dropoff.

This is the action space : the set of all actions that our agent can take in a given state.

17. Let’s started. We can load the game environment and render what it looks like



```python
import gym
env = gym.make("Taxi-v3").env
env.render()
```



```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

18. Here’s our restructured problem statement : “There are 4 location (labeled by different letters), and our job is to pick up the passenger at one location and drop him/her off at another. We receive +20 points for a successful drop-off and lose 1 point every time-step it takes. There is also a 10 point penalty for illegal pick up and drop off actions.”
Let’s dive more into the environment.

```python
env.reset() # reset environment to a new, random state
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))
```

```
+---------+
|R: | : :G|
| : | : : |
| : : :█: |
| | : | : |
|Y| : |B: |
+---------+

Action Space Discrete(6)
State Space Discrete(500)
```

- The **filled square** represents the taxi, which is yellow without passenger and green with a passenger
- The **pipe (“|”)** represents a wall which the taxi cannot cross
- **R, G, Y, B** are the possible pickup destination location. The **blue letter** represents the current passenger pickup location, and **the purple letter** is the current destination.

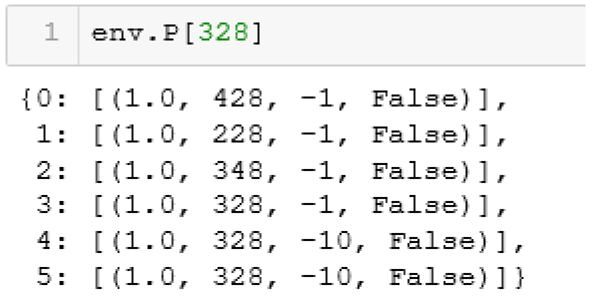
19. We can actually take our illustration above, encode its state, and give it to the environment to render in Gym. Recall that we have the taxi at row 3, column 1, our passenger is at location 2 and our destination is location 0. Using the Taxi-v3 state encoding method, we can do the following :

```python
# (taxi row, taxi column, passenger index, destination index)
state = env.encode(3, 1, 2, 0)
print("State:", state)

env.s = state
env.render()
```

```
State: 328
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| |█: | : |
|Y| : |B: |
+---------+
```

We are using our illustration’s coordinates to generate a number corresponding to a state between 0 and 499, which turns out to be **328** for our illustration’s.

20. When the Taxi environment is created, there is an initial Reward table that's also created, called 'P'. We can think of it like a matrix that has the number of states as rows and number of actions as columns, i.e : $states \times actions$ matrix.
Since every state is in this matrix, we can see the default reward values assigned to our illustration's state :



```
env.P[328]
```

```
{0: [(1.0, 428, -1, False)],
 1: [(1.0, 228, -1, False)],
 2: [(1.0, 348, -1, False)],
 3: [(1.0, 328, -1, False)],
 4: [(1.0, 328, -10, False)],
 5: [(1.0, 328, -10, False)]}
```

The dictionary has the structure **{ action : [(probability, nextstate, reward, done)] }**.

A few things to note :

- The 0-5 corresponds to the actions **(south, north, east, west, pickup, dropoff)** the taxi can perform at our current state in the illustration.
- In this env, probability is always **1.0**.
- The **nextstate** is the state we would be in if we take the action at this index of the dict
- All the movement action have a -1 reward and the pickup/dropoff actions have -10 reward in this particular state. If we are in a state where the taxi has a passenger and is on top of the right destination, we would see a reward of 20 at the dropoff action (5).
- **Done** is use to tell us when we have successfully dropped off passenger in the right location. Each successful dropoff is the end of an **episode**.

<u>SARSA (STATE-ACTION-REWARD-STATE-ACTION) ALGORITHM</u>

SARSA algorithm is a slight variation of the Q-Learning algorithm. For a learning agent in any Reinforcement Learning algorithm it's policy can be of two types :

a. On – Policy : in this, the learning agent learns the value function according to the current action derived from the policy currently being used
b. Off – Policy : in this, the learning agent learns the value function according to the action derived from another policy.

The Q-learning algorithm is an Off Policy technique. On the other hand, SARSA technique is an On Policy and uses the action performed by the current policy to learn the Q-value.

The following is python code demonstrates how to implement the SARSA algorithm using the OpenAI gtm module to load the environment.

21. Importing the required libraries

```python
import numpy as np
import gym
```

22. Building the environment

Here, we will be using the ‘FrozenLake-v0’ environment which is preloaded into gym.

```python
#Building the environment
env = gym.make('FrozenLake-v0')
```

23. Initializing different parameters

```python
#Defining the different parameters
epsilon = 0.9
total_episodes = 10000
max_steps = 100
alpha = 0.85
gamma = 0.95

#Initializing the Q-matrix
Q = np.zeros((env.observation_space.n, env.action_space.n))
```

24. Defining utility function to be used in the learning process

```python
#Function to choose the next action
def choose_action(state):
    action=0
    if np.random.uniform(0, 1) < epsilon:
        action = env.action_space.sample()
    else:
        action = np.argmax(Q[state, :])
    return action

#Function to learn the Q-value
def update(state, state2, reward, action, action2):
    predict = Q[state, action]
    target = reward + gamma * Q[state2, action2]
    Q[state, action] = Q[state, action] + alpha * (target - predict)
```

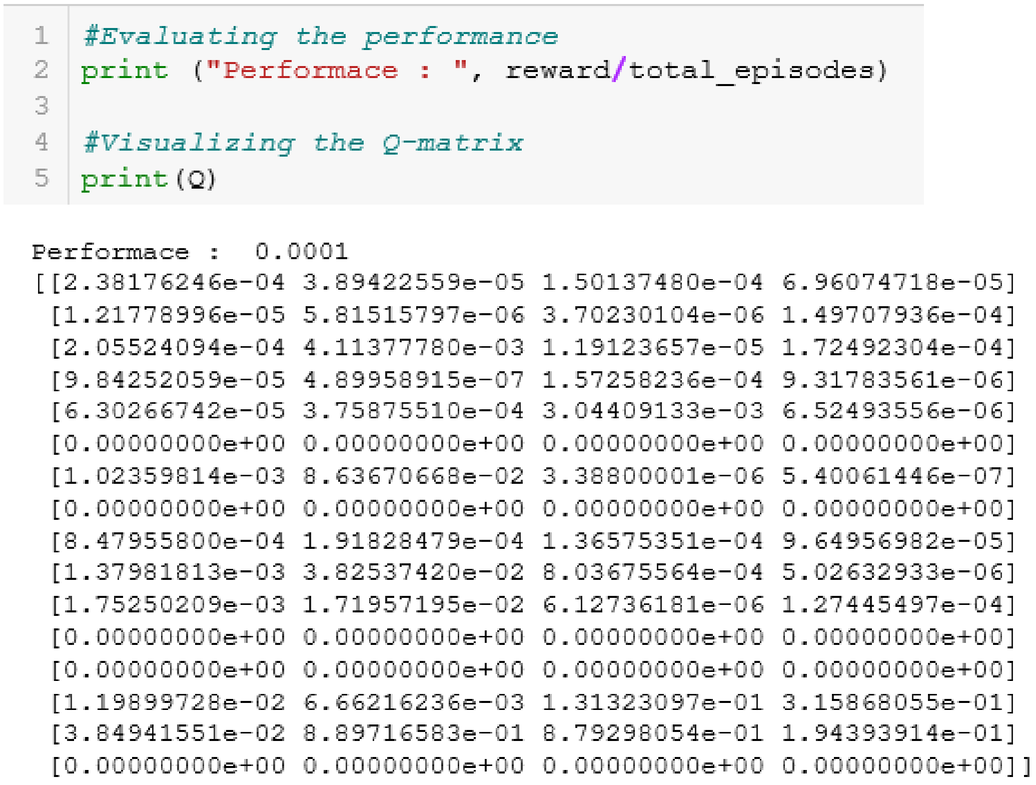
25. Training the learning agent

```python
#Initializing the reward
reward=0

# Starting the SARSA learning
for episode in range(total_episodes):
    t = 0
    state1 = env.reset()
    action1 = choose_action(state1)

    while t < max_steps:
        #Visualizing the training
        env.render()

        #Getting the next state
        state2, reward, done, info = env.step(action1)

        #Choosing the next action
        action2 = choose_action(state2)

        #Learning the Q-value
        update(state1, state2, reward, action1, action2)

        state1 = state2
        action1 = action2

        #Updating the respective vaLues
        t += 1
        reward += 1

        #If at the end of learning process
        if done:
            break
```

```
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
```

In the above output, the red mark determines the current position of the agent in the environment while the direction given in brackets gives the direction of movement that the agent will make next. Note that the agent stays at it's position if goes out of bounds.

26. Evaluating the performance



```
#Evaluating the performance
print ("Performace : ", reward/total_episodes)

#Visualizing the Q-matrix
print(Q)
```

```
Performace :  0.0001
[[2.38176246e-04 3.89422559e-05 1.50137480e-04 6.96074718e-05]
 [1.21778996e-05 5.81515797e-06 3.70230104e-06 1.49707936e-04]
 [2.05524094e-04 4.11377780e-03 1.19123657e-05 1.72492304e-04]
 [9.84252059e-05 4.89958915e-07 1.57258236e-04 9.31783561e-06]
 [6.30266742e-05 3.75875510e-04 3.04409133e-03 6.52493556e-06]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.02359814e-03 8.63670668e-02 3.38800001e-06 5.40061446e-07]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [8.47955800e-04 1.91828479e-04 1.36575351e-04 9.64956982e-05]
 [1.37981813e-03 3.82537420e-02 8.03675564e-04 5.02632933e-06]
 [1.75250209e-03 1.71957195e-02 6.12736181e-06 1.27445497e-04]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [1.19899728e-02 6.66216236e-03 1.31323097e-01 3.15868055e-01]
 [3.84941551e-02 8.89716583e-01 8.79298054e-01 1.94393914e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
```

Referensi :

Van derPlas J. 2016. Pyton Data Science Handbook. Oreilly Media Inc