Implement a basic driving agent:

For the first part of this assignment, Agent1.py is the respective file for that task. It contains code for processing next waypoint, intersections states captured via the input array, and current deadline value. And as required, I produced random action, which the agent later will ultimately learn. All I have done for this part of task is generated random index for the probable actions and assign that task to the agent to proceed using the below code

```
probable_actions = Environment.valid_actions

random_action = random.randint(0,len(probable_actions)-1)

action = probable_actions[random_action]
```

Then I make the agent run with enforce_deadline set to false. It moves randomly, but after taking random steps, it reaches the goal(but it was entirely by chance). Next time I run that it have not converged for next 3 minutes.

Identify and update state:

Initially, I was thinking of taking following variables as input states,

- inputs['light']
- self.next_waypoint
- int(inputs['oncoming']=='right') # for learning right oncoming traffic rule in case of red light
- int(inputs['oncoming']=='left') # for learning right oncoming traffic rule in case of red light

But after training the smart car with QLearner, I updated them to just first two of them. This is because considering all of the possible scenarios, 2 for the light variable, 3 for a waypoint, 2 for each of last two variables, it will make a total of 24 and when computed with four possible actions it will make total entries 96. So, I removed the last two of variables and then using the total of 24 cases, the model will learn pretty quickly. And the second reason for that might be considering a small number of cars, the probability of use cases where the light will be red and either oncoming will be right or left will be pretty small and the stater code donot incur penalties to the agent for interacting with cars or right of the way. So, I used just first two of them as the state variable for a model.

Other input parameters include the deadline, which I also do not select to include it in the state variable because of its large number of discrete values. For instance, its values vary between 0-55(normally which I observed) and including this as parameter to the state will make total entries of 24*56 = 1344 for the QLearner dictionary, which will take a lot of time to train the model and the issue of curse of dimensionality arise i.e. we need at least as many examples as necessary to cover all the variations of configurations of interest, in order to produce a correct answer around each of these configurations. And it is still possible for us to learn well from a relatively small number of examples by using a small number of features as long as apparent complexity is actually captured by those features. So, for making model simple as favored by Occam Razor and reducing learning time, I have not included that into state variable.

These two of them will capture most of the things that an agent need to learn for successfully navigating through that environment. As from light variable, it will learn to stop on the red light. And for the next_waypoint variable, it will learn to take appropriate action when certain waypoint comes. So, these two of them are more than enough for modeling the agent and its environment.

Implement Q-Learning:

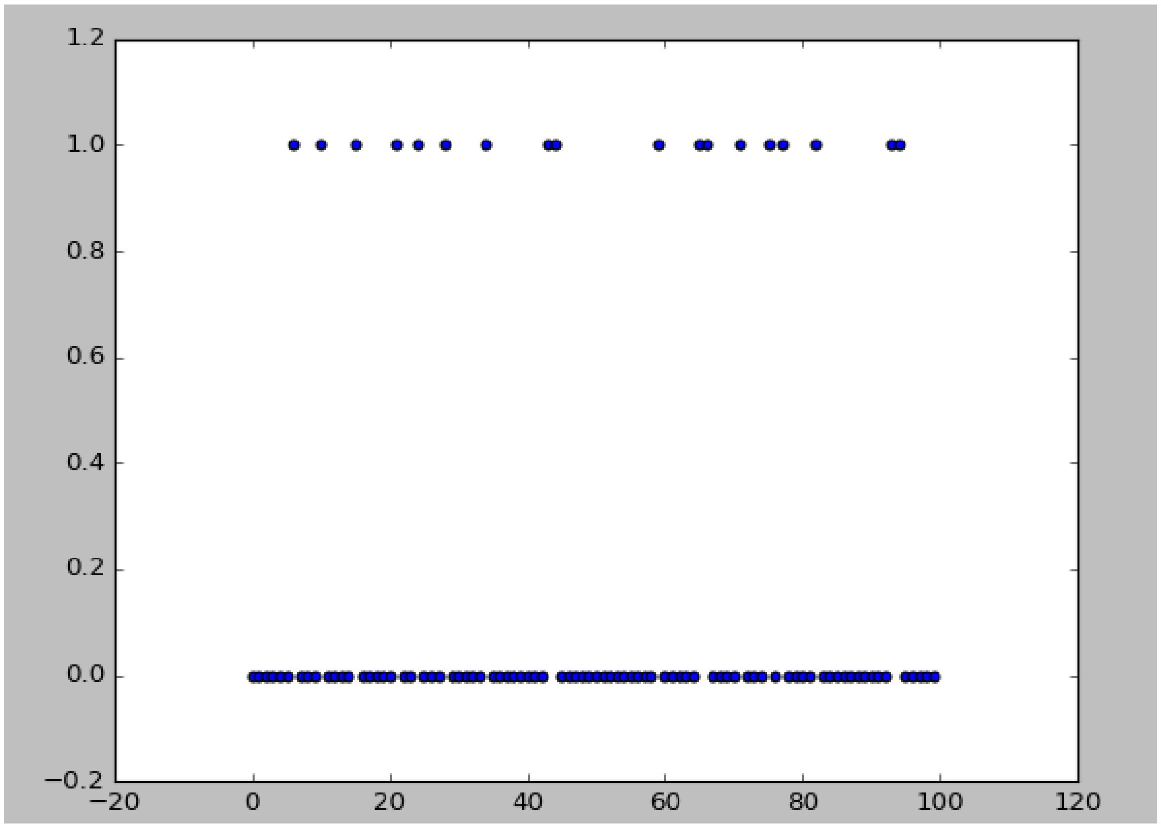
I implemented the QLearning algorithm in a separate class named QLearn.py. In that class, firstly I initialized all of the necessary variables needed for QLearning algorithm at the init of that class or constructor of that class. I stored Q values in the dictionary with the key (l_state,l_action). As it will handle all of the combinations of states with actions, so it can be used for carrying further calculations. Then I used get_Q_value to get Q value at any actions and state combination, and check there if the key is present there then it will return its value, else it will add default_value for that key and return that.

Then, I make update_Q method for updating the value of Q given any combination of (previous_state,action,reward,new_state). In the SmartAgent.py file, I used initial state and action variable for storing previous state. For the first iteration, it will select a random action from Get_action method and then for later iterations, it will update its values correspondingly.

Then, I made Get_action method for getting appropriate action based on the Q values computed for that state, using all of the actions possible. If there exist ties between some actions subset, then it will randomly select any actions from that action subset and proceed further.

Running agent with Q-Learn algorithm:

After implementing necessary code in the SmartAgent.py file for that QLearn class, I further implemented plot_trials method that will plot the trials of the smart car for each iteration and whether it was successful or failed. I exploited following condition for checking that whether the deadline is zero and the reward for that state is less than 10, then it must be categorized in the failure state. Else the agent has accomplished the goal. So, I checked that condition for each iteration of update method of the learningAgent class and used last iteration value of that trial. So, using default settings like 0 for learning rate, 0 for discounting factor and zero for default initial value of Q, I get below plot for 100 trials of smart car,



So, the QLearning algorithm has not performed pretty well for learning the route. This is because of not tuning parameters for the QLearning algorithm. In the next section, I have tuned various parameters for getting optimal performance.

Enhance the driving agent:

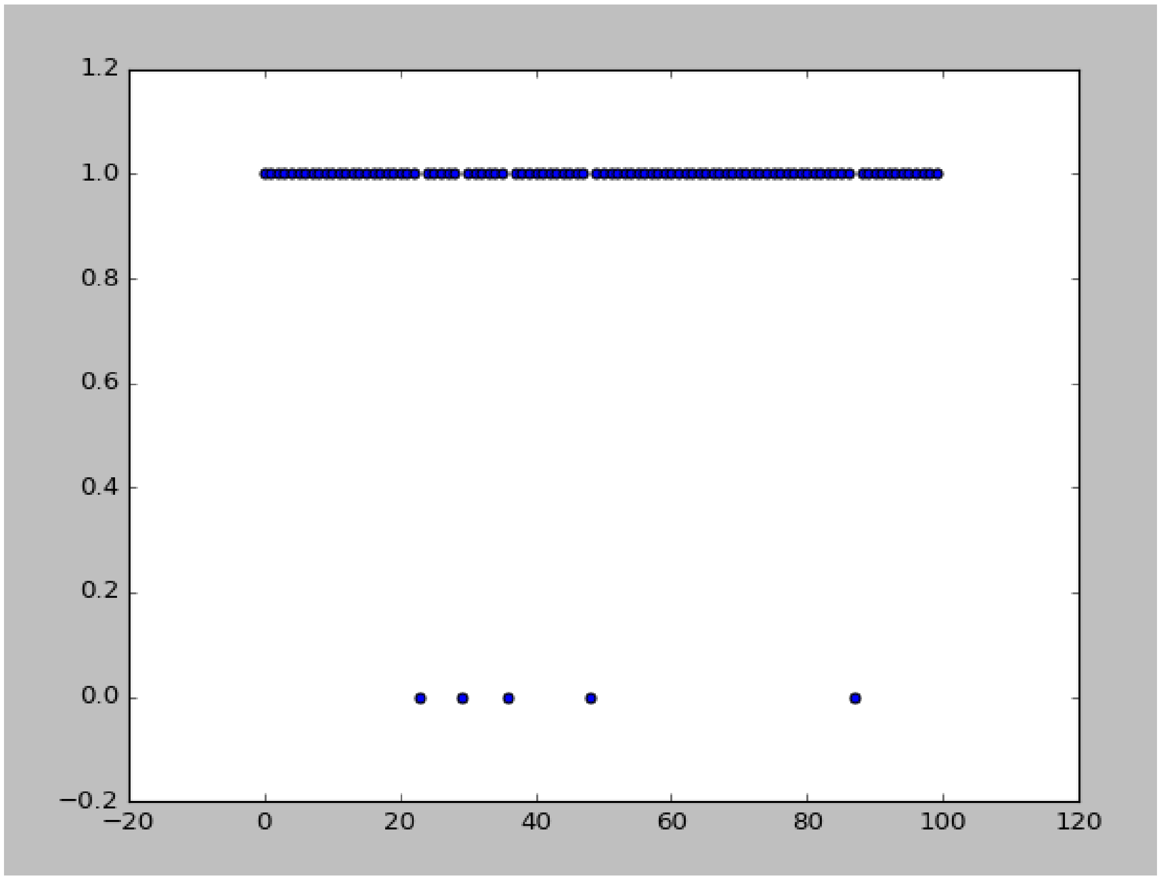
As seen before, the algorithm has not performed pretty well for learning environment. So, I changed parameters of that algorithm to these values,

```
self.learning_rate = 0.5

self.discounting_factor = 0.5

self.default_val = 0
```

and got the pretty good performance of the agent. It successfully learnt the environment and got accuracy of 10/10 in the last 10 iterations.

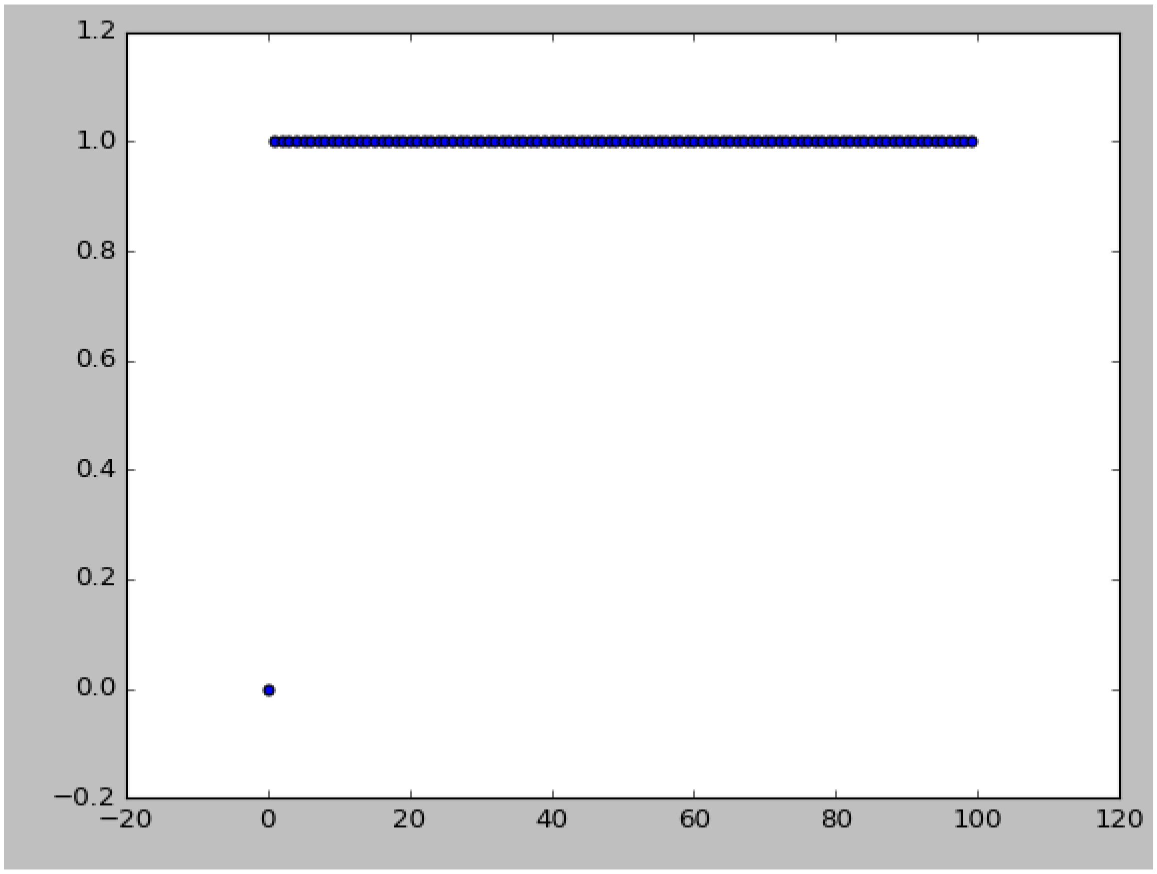


Now tweaking further parameters for getting optimal performance, I tried different value, but get optimal performance on below value,

```
self.learning_rate = 0.5

self.discounting_factor = 0.3

self.default_val = 2
```



The reason for this choice of values is coherent with theory. As learning rate will determine which extant the algorithm will use newly captured information as compared to its old value. So, using the moderate value of learning rate will make QLearning utilize both of the old and new value combination to come up with the better decision and utilize both of the newly captured information in conjunction with its old value.

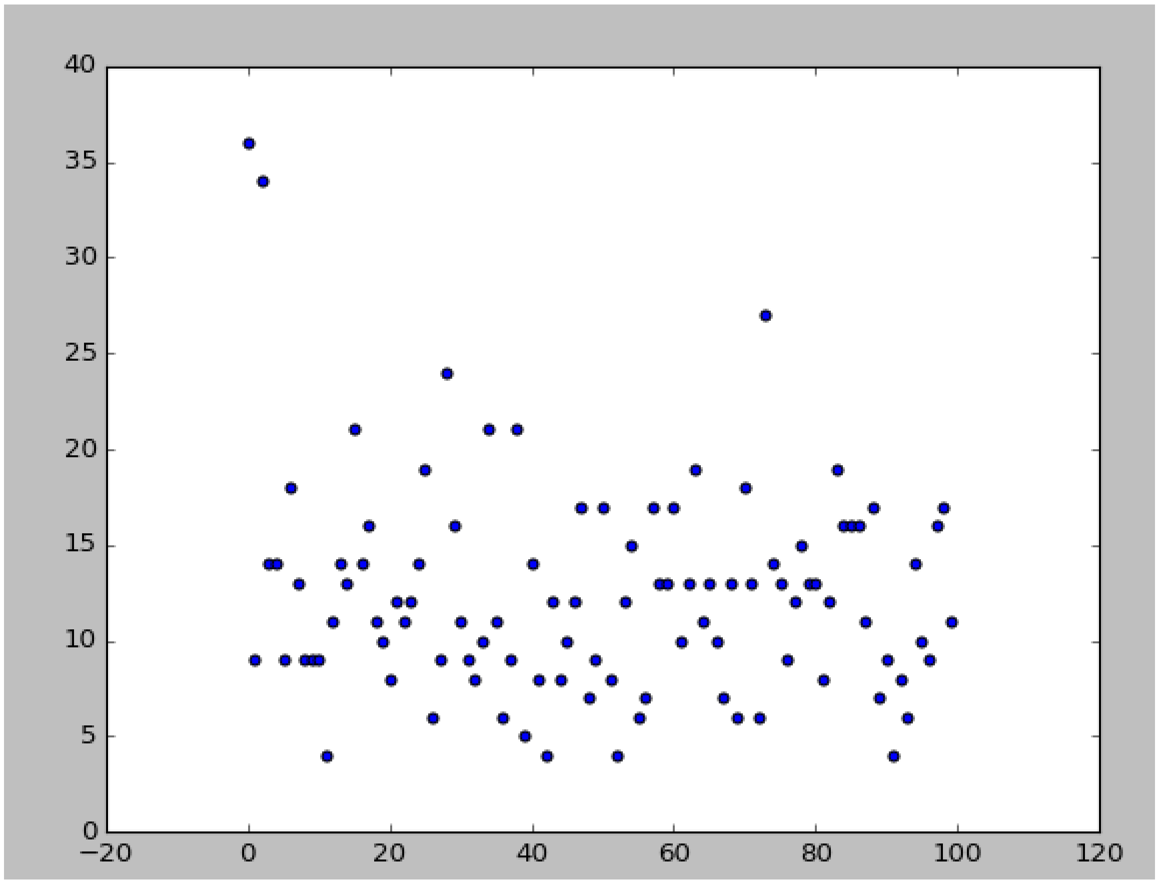
For the discounting factor , it will tell the significance of future delayed reward. As using 0.3 value of it will make an algorithm give significance to the delayed reward as compared to reward to that state.

For the initial condition initialised to 2, it will allow the model to do more exploration as compared to small initial values, and hight initial values which are also called optimal initial conditions can enforce model to do more exploration.

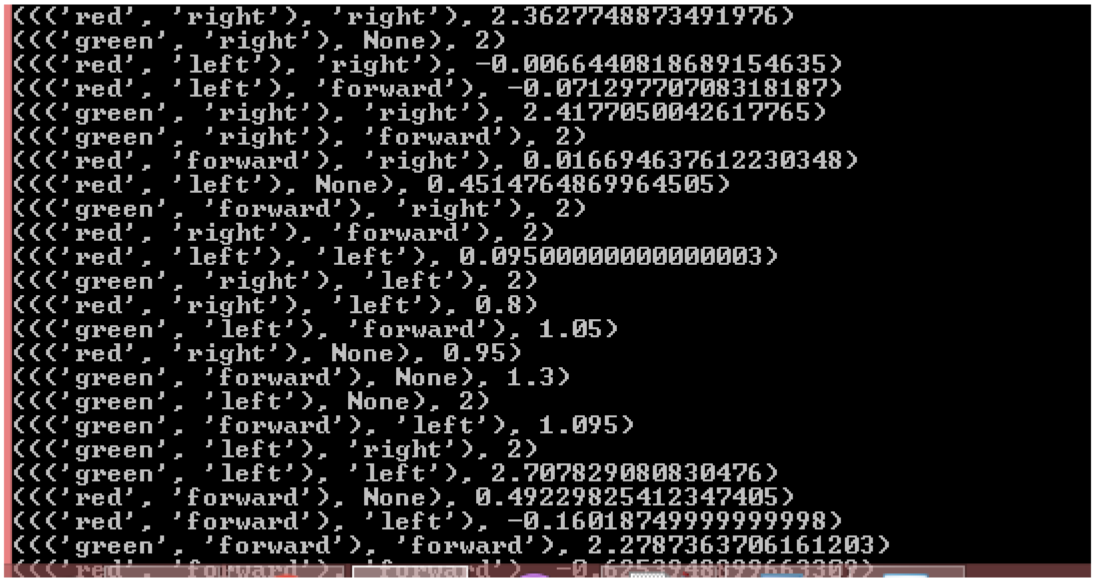
The above graphs is plotted with trial result on y-axis i.e. 1 for successful completion and 0 for failed to reach there along with number of trials on the x-axis.

Now coming to the last question that **whether your agent gets close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties**?

As below graph indicates steps taken by the algorithm for reaching to its destination plotted on the y-axis against a number of trials, which indicates that in general as the QLearner will take approximately 10-20 steps to reach its destination. At the start, it explores the environment, but with the passage of trials, the number of steps taken by it to reach destination reaches to 10-20 depending upon route given by planner.



Other proof will be considering below Q values of the resultant QLearning algorithm after 100 trials, it indicates that the algorithm assigns high values to the waypoint versus corresponding actions and to the light variable correspondingly (for example for the green/right state it has assigned more weight as compared to other actions) . So, the algorithm will ultimately reach to optimal solution.



```
((('red', 'right'), 'right'), 2.3627748873491976)
((('green', 'right'), None), 2)
((('red', 'left'), 'right'), -0.0066440818689154635)
((('red', 'left'), 'forward'), -0.07129770708318187)
((('green', 'right'), 'right'), 2.4177050042617765)
((('green', 'right'), 'forward'), 2)
((('red', 'forward'), 'right'), 0.016694637612230348)
((('red', 'left'), None), 0.4514764869964505)
((('green', 'forward'), 'right'), 2)
((('red', 'right'), 'forward'), 2)
((('red', 'left'), 'left'), 0.09500000000000003)
((('green', 'right'), 'left'), 2)
((('red', 'right'), 'left'), 0.8)
((('green', 'left'), 'forward'), 1.05)
((('red', 'right'), None), 0.95)
((('green', 'forward'), None), 1.3)
((('green', 'left'), None), 2)
((('green', 'forward'), 'left'), 1.095)
((('green', 'left'), 'right'), 2)
((('green', 'left'), 'left'), 2.707829080830476)
((('red', 'forward'), None), 0.49229825412347405)
((('red', 'forward'), 'left'), -0.16018749999999998)
((('green', 'forward'), 'forward'), 2.2787363706161203)
((('red', 'forward'), 'forward'), -0.62[illegible])
```

I made below settings for getting results faster by changing not to display and update_delay to very small values in the final code.

```
self.update_delay=0

self.display = False
```