



UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

INTELLIGENT SYSTEM

ASSIGNMENT#5

MOHAMMAD HEYDARI

810197494

UNDER SUPERVISION OF:

DR. RESHAD HOSSINI

ASSISTANT PROFESSOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

Jan. 2022

1 CONTENTS

2	Question #1	3
2.1	Naïve Bayes	3
3	Question #2	8
3.1	Model-Based Reinforcement Learning (Python Implementing).....	8
3.2	Model-Based Reinforcement Learning(Theoretical)	16
4	Question #3	20
4.1	Un-Model-Based Reinforcement Learning(Q-Learning)	20
	Hyper-parameters and optimizations:	21
5	Acknowledgement	34
6	References.....	35

2 QUESTION #1

2.1 NAÏVE BAYES

In this part we intend to implement Naïve Bayes algorithm in order to classify the Mushroom data.

This dataset consists of two .csv files, one of them refers to train-data and another one refers to test-data.

When I saw the train-data I found that there was a feather called **stalk-root** which had unknown value, so in this case I decided to ignore them in step of calculating likelihood probability and I have only counted the number of deterministic characters.

Note that in the test-data we also face with '?' character but according to the algorithm the trick is to pass from columns which have these value because we don't care about the value of unknown characters and just obtained the probability of other columns and at the final we multiply the calculated probability.

First of all, let's have a look on Naïve Bayes concept and how it can be implement.

Naïve Bayes algorithm is a supervised classification algorithm based on Bayes theorem with strong(Naïve) **independence** among features.

In probability theory and statistics, Bayes' theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

Naive Bayes Classifier formula can be written based on Bayes theorem as:

$$P(y | x_1, \dots, x_j) = \frac{P(x_1, \dots, x_j | y)P(y)}{P(x_1, \dots, x_j)}$$

Naive Bayes Classifier Formula

Where,

x_1, \dots, x_j are j features that are independent of each other. y is the dependent variable.

$P(y | x_1, \dots, x_j)$: Posterior Probability

$P(x_1, \dots, x_j | y)$: Likelihood of features x_1 to x_j given that their class is y .

$P(y)$: Prior Probability

$P(x_1, \dots, x_j)$: Marginal Probability

Now, it's time to dive into the codes:

Here you can find all of my implementation in this regards:

First of all, I attach the whole parts of code and afterward I have provided some tips about the functions and implementation.

Naive Bayes Algorithm

```
import numpy as np
import pandas as pd
```

```
df_train = pd.read_csv('Mushroom_Train.csv')
df_test = pd.read_csv('Mushroom_Test.csv')
```

```
X_train=df_train.drop(['class'], axis = 1)
y_train=df_train['class']
X_test=df_test.drop(['class'], axis = 1)
y_test=df_test['class']
```

```
featureDict = {}
count=0
for col_name in X_train.columns:
    featureDict[count]= np.unique(X_train[col_name])
    count=count+1
```

```
def calculate_confusion(actual, predicted):

    classes = np.unique(actual)
    confusion_matrix = np.zeros((len(classes), len(classes)))
    for i in range(len(classes)):
        for j in range(len(classes)):
            confusion_matrix[i, j] = np.sum((actual == classes[i]) & (predicted == classes[j]))

    return confusion_matrix
```

```
def Prior_Probability_Calculator(df_train):
    target=df_train["class"]
    e_count=sum(target == 'e')
    p_count=sum(target == 'p')
    return e_count/target.size,p_count/target.size
```

```
def Likelihood_Probability_Calculator(X_train,y_train):

    e_count=sum(y_train == 'e')
    p_count=sum(y_train == 'p')

    train_e = X_train[y_train=='e']
    train_p = X_train[y_train=='p']

    dictionary_probE = {}
    dictionary_probP = {}
    cnt=0
    for col_name in train_e.columns:

        e_unique, e_counts = np.unique(train_e[col_name],return_counts=True)
        p_unique, p_counts = np.unique(train_p[col_name],return_counts=True)
        e_countDict = dict(zip(e_unique,e_counts))
        p_countDict = dict(zip(p_unique,p_counts))
        prob_e = {}
        prob_p = {}
        for st in featureDict[cnt]:
            if st in e_countDict:
                prob_e[st]=float(e_countDict[st]/e_count)
            else:
                prob_e[st]=float(0)

            if st in p_countDict:
                prob_p[st]=float(p_countDict[st]/p_count)
            else:
                prob_p[st]=float(0)
```

```

dictionary_probE[cnt]=prob_e
dictionary_probP[cnt]=prob_p
cnt=cnt+1
return dictionary_probE,dictionary_probP

```

```

def predict(X_test,e_prior,p_prior):

    length,d=np.shape(X_test)

    stored_data = np.zeros((length,2), dtype=float)

    cnt2=0
    for idx,row in X_test.iterrows():
        eProb=e_prior
        pProb=p_prior
        for j in range(0,d):

            featuree=row[j]
            if featuree !='?':
                eProb = eProb * featureGivenEd[j][featuree]
                pProb = pProb * featureGivenPo[j][featuree]
            stored_data[cnt2][0]=eProb
            stored_data[cnt2][1]=pProb
            cnt2=cnt2+1

    predicted = np.zeros((length), dtype='str')
    for i in range(0,length):
        if np.argmax(stored_data[i]) == 0:
            predicted[i]='e'
        else:
            predicted[i]='p'
    return predicted

```

```

e_prior,p_prior=Prior_Probability_Calculator(df_train)
dictionary_probE,dictionary_probP=Likelihood_Probability_Calculator(X_train,y_train)
y_test_pred=predict(X_test,e_prior,p_prior)
L,dd=np.shape(X_test)

print ("Accuracy percent is:")
print ("%",(y_test_pred==y_test).sum()*100/L)

```

```

Accuracy percent is:
% 99.23664122137404

```

```

confusion_matrix=calculate_confusion(y_test,y_test_pred)

print("confusion Matrix is:\n")
print(confusion_matrix.astype(int))

```

```

confusion Matrix is:

```

```

[[245  4]
 [ 0 275]]

```

Considorable notes:

In this implementation I have used four function, the first one `confusion-matrix()` is implemented to calculate the confusion matrix from scratch, and the second one counts the number of 'p' and 'e' character and then by getting the size of y-train in can be give us the Prior Probability, afterward according to the notation which discussed in previous section we are going to calculate and store the Likelihood table for all features by making the likihood-probability calculator.

If I want to add more detailed notes about the implementation of this function I want to point to the most important command which I have used and that's the `unique()` function which let me to access much easier to the unique character and also their counts so going through this way I have made a dictionary which store the probability of each character in each column of our data.

Now with training the model according to the probability calculation we are supposed to predict the label of each single rows in the test-data.csv in this way I have provided the predict() function which uses the x-test data and also prior probability to make a prediction in our case.

In other word, we pass through the rows and when face with a specific character turn into our probability matrix and take the correspond probability and then we multiply the current one with the previous using some simple syntax in my codes.

One of the other important tips I want to issue here is about the unknown character in the test data, in this regard I have easily ignored these values by writing a if which never let unknown data ('?') to have chance of affecting on our results.

Finally I have instantiate the mentioned function in a new cell and running all together keeps me filling happy when I reach to the correct and resonable results as you can see :

```
e_prior,p_prior=Prior_Probability_Calculator(df_train)
dictionary_probE,dictionary_probP=Likelihood_Probability_Calculator(X_train,y_train)
y_test_pred=predict(X_test,e_prior,p_prior)
L,dd=np.shape(X_test)

print ("Accuracy percent is:")
print ("%",(y_test_pred==y_test).sum()*100/L)
```

```
Accuracy percent is:
% 99.23664122137404
```

```
confusion_matrix=calculate_confusion(y_test,y_test_pred)

print("confusion Matrix is:\n")
print(confusion_matrix.astype(int))
```

```
confusion Matrix is:
```

```
[[245  4]
 [ 0 275]]
```

As you can see we have reached to 99.2% accuracy which is so nice and so exciting in our case.

Interpreting our findings:

As we know A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class.

This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by our classifier but more importantly the types of errors that are being made. It is this breakdown that overcomes the limitation of using classification accuracy alone.

“true positive” for correctly predicted event values, and in our case it’s **245**

“false positive” for incorrectly predicted event values, and in our case it’s **4**

“true negative” for correctly predicted no-event values, and in our case it’s **0**

“false negative” for incorrectly predicted no-event values, and in our case it’s **275**

Confusion matrix abbreviated below:

$$\begin{bmatrix} TP = 245 & TN = 4 \\ FP = 0 & FN = 275 \end{bmatrix}$$


And at the end, Accuracy is one metric for evaluating classification models. Informally, accuracy is the fraction of predictions our model got right. Formally, accuracy has the following definition:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

$$\text{Accuracy} = \frac{245+275}{245+275+0+4} = \frac{520}{524} = 0.922 = 92.2\%$$

Where TP = True Positives, TN = True Negatives, FP = False Positives, and FN = False Negatives.

For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them


3 QUESTION #2

3.1 MODEL-BASED REINFORCEMENT LEARNING (PYTHON IMPLEMENTING)

In this part we intend to implement a problem using model-based reinforcement learning (policy iteration algorithm) and going through this way we reach to an optimal policy which show us the capacity transmission between two company.

First of all, let's have a dive into the policy iteration algorithm as of our approach to solve the problem:

Policy Iteration is an algorithm in Reinforcement Learning, which helps in learning the optimal policy which maximizes the long term discounted reward. These techniques are often useful, when there are multiple options to choose from, and each option has its own rewards and risks.

Policy Iteration steps:

1) Randomly initialize the policy. Initialize actions randomly at every state of the system.

2) Step 2 is based on Bellman's equation which is provided below:

$$V(s) = r(s) + \gamma * \max_{s', r} (\sum p(s', r | s, \pi(s)) * V(s'))$$

Policy Evaluation¹

Get action for every state in the policy and evaluate the value function using the above equation. Here is p is the transition probability, also denoted by T.

3) Policy Improvement

For every state, get the best action from value function using.

$$\pi(s) = \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) * V(s')$$

Policy Improvement¹

What is a Policy:

Policy is a mapping of an action to every possible state in the system. An optimal policy is that policy which maximizes the long term reward. Thus, goal is to find that optimum policy.

Policy Iterations:

Iterate steps 2 and 3, until convergence. If the policy did not change throughout an iteration, then we can consider that the algorithm has converged.

Now, let's first attach the codes and after have a more detailed information about my implementation:

Here you can find all of my code implementation using Python if there is any issue with that please let me know to provide more detailed information. (be in touch [here!](#))

Policy Iteration

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats import poisson
```

```
MAX_MOVE= 5
REQUEST_FIRST = 3
REQUEST_SECOND= 4
RETURNS_FIRST = 3
RETURNS_SECOND= 2
CREDIT = 10
```

```
def curve(iterations,policy,max_possible_capacity):
    fig = sns.heatmap(np.flipud(policy), cmap="YlGnBu", ax=axes[iterations])
    fig.set_ylabel('# capacity at first location', fontsize=30)
    fig.set_yticks(list(reversed(range(max_possible_capacity + 1))))
    fig.set_xlabel('# capacity at second location', fontsize=30)
    fig.set_title('policy {}'.format(iterations), fontsize=30)
```

```
def optimal_curve(iterations,value,max_possible_capacity):
    fig = sns.heatmap(np.flipud(value), cmap="YlGnBu", ax=axes[-1])
    fig.set_ylabel('# capacity at first location', fontsize=30)
    fig.set_yticks(list(reversed(range(max_possible_capacity + 1))))
    fig.set_xlabel('# capacity at second location', fontsize=30)
    fig.set_title('optimal value', fontsize=30)
```

```
poisson_cache = dict()
def poisson_probability(n, lam):
    global poisson_cache
    key = n * 10 + lam
    if key not in poisson_cache:
        poisson_cache[key] = poisson.pmf(n, lam)
    return poisson_cache[key]

def expected_return(state,action,state_value,gamma,purchase_cost):
    value = 0
    value -= purchase_cost * abs(action)
    sta_num1 = min(state[0] - action,20)
    sta_num2 = min(state[1] + action,20)
    for request_first in range(11):
        for request_second in range(11):
            prob = poisson_probability(request_first,REQUEST_FIRST) * poisson_probability(request_second,REQUEST_SECOND)
            vid_first = min(sta_num1,request_first)
            vid_second = min(sta_num2,request_second)
            reward = (vid_first + vid_second) * CREDIT
            sta_num1 = sta_num1-vid_first
            sta_num2 = sta_num2-vid_second
            returned_first = RETURNS_FIRST
            returned_second = RETURNS_SECOND
            sta_num1 = min(sta_num1 + returned_first, 20)
            sta_num2 = min(sta_num2 + returned_second,20)
            value += prob * (reward + gamma * state_value[sta_num1,sta_num2])
    return value
```

```
def policy_evaluation(initial_value,policy,gamma,purchase_cost):
    while True:
        old_value = initial_value.copy()
        for i in range(21):
            for j in range(21):
                state=[i, j]
                value[i, j] = expected_return(state,policy[i, j],initial_value,gamma,purchase_cost)
            max_value_change = abs(old_value - value).max()
            print('max value change {}'.format(max_value_change))
            if max_value_change < 1e-4:
                break
        return value

def policy_improvement(value,gamma,purchase_cost):
    pi_holder=[]
    new_policy=np.zeros([21,21])
    for i in range(21):
        for j in range(21):
            state=[i,j]
            action_array=[-5,-4,-3,-2,-1,0,1,2,3,4,5]
            for k in action_array:
                pi=expected_return(state,k,value,gamma,purchase_cost)
                pi_holder.append(pi)
            max_val=max(pi_holder)
            max_index = pi_holder.index(max_val)
            new_policy[i,j]=action_array[max_index]
            pi_holder=[]
    return new_policy
```

Discount factor == 0.9

```
gamma=0.9
MAX_capacity=20
purchase_cost=2
value = np.zeros((MAX_capacity + 1, MAX_capacity + 1))
policy = np.zeros(value.shape, dtype=np.int)
iterations = 0
_, axes = plt.subplots(2, 3, figsize=(40, 20))
plt.subplots_adjust(wspace=0.1, hspace=0.2)
axes = axes.flatten()
while True:
    curve(iterations,policy,20)
    policy=policy.astype(int)
    old_policy=policy.copy()
    value=policy_evaluation(value,policy,gamma,purchase_cost)
    policy=policy_improvement(value,gamma,purchase_cost)
    policy=policy.astype(int)
    if np.array_equal(old_policy,policy)==True:
        optimal_curve(iterations,value,20)
        break
    iterations += 1
plt.show()
print(policy)
```

Changing the Purchase Cost == 6

```
gamma=0.9
purchase_cost=6
MAX_capacity=20
value = np.zeros((MAX_capacity + 1, MAX_capacity + 1))
policy = np.zeros(value.shape, dtype=np.int)
iterations = 0
_, axes = plt.subplots(2, 3, figsize=(40, 20))
plt.subplots_adjust(wspace=0.1, hspace=0.2)
axes = axes.flatten()
while True:
    curve(iterations,policy,20)
    policy=policy.astype(int)
    old_policy=policy.copy()
    value=policy_evaluation(value,policy,gamma,purchase_cost)
    policy=policy_improvement(value,gamma,purchase_cost)
    policy=policy.astype(int)
    if np.array_equal(old_policy,policy)==True:
        optimal_curve(iterations,value,20)
        break
    iterations += 1
plt.show()
print(policy)
```

Changing the Discount factor == 1

```

gamma=1
purchase_cost=2
MAX_capacity=20
value = np.zeros((MAX_capacity + 1, MAX_capacity + 1))
policy = np.zeros(value.shape, dtype=np.int)
iterations = 0
_, axes = plt.subplots(2, 3, figsize=(40, 20))
plt.subplots_adjust(wspace=0.1, hspace=0.2)
axes = axes.flatten()
while True:
    curve(iterations,policy,20)
    policy=policy.astype(int)
    old_policy=policy.copy()
    value=policy_evaluation(value,policy,gamma,purchase_cost)
    policy=policy_improvement(value,gamma,purchase_cost)
    policy=policy.astype(int)
    if np.array_equal(old_policy,policy)==True:
        optimal_curve(iterations,value,20)
        break
    iterations += 1
plt.show()
print(policy)

```

As you can see in this part I implement my code using several functions which working together and at the final get us the result.

First of all, I have written a function which calculate the Poisson probability based on its equation, furthermore I have implemented expected-reward function which calculate the transition reward after that I have written the evaluation policy function based on bellman equation which gives us the new value function just like the process which we have passed in theoretical part.

Furthermore, we have implemented improvement policy function which takes the new value function and actions and get us the improved policy and at the final step I have called my function in different cells to show each result according to each part!

Further you can see the results of my code:

For running this part please have an accurate look on related directory and There you can easily find all of you need and also I would appreciate it if you would consider them



the Discount Factor equals to 0.9:

Further you can see the result:

```
[[ 0  0  0  0  0  0  0  0  0 -1 -1 -2 -2 -2 -3 -3 -3 -3 -3 -4 -4 -4]
 [ 0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -2 -2 -2 -2 -2 -3 -3 -3]
 [ 0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1 -1 -2 -2 -2 -2]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1 -1 -1 -1 -1]
 [ 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 -1]
 [ 1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 2  2  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 3  2  2  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 3  3  3  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 4  4  3  2  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  4  3  3  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  4  4  3  2  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  4  3  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  4  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  2  2  1  1  1  1  1  1  1  0  0  0  0  0  0  0]
 [ 5  5  5  4  3  3  2  2  2  2  2  2  2  1  1  1  1  1  0  0  0]]
```

Figure 1: Representation of optimum Policy

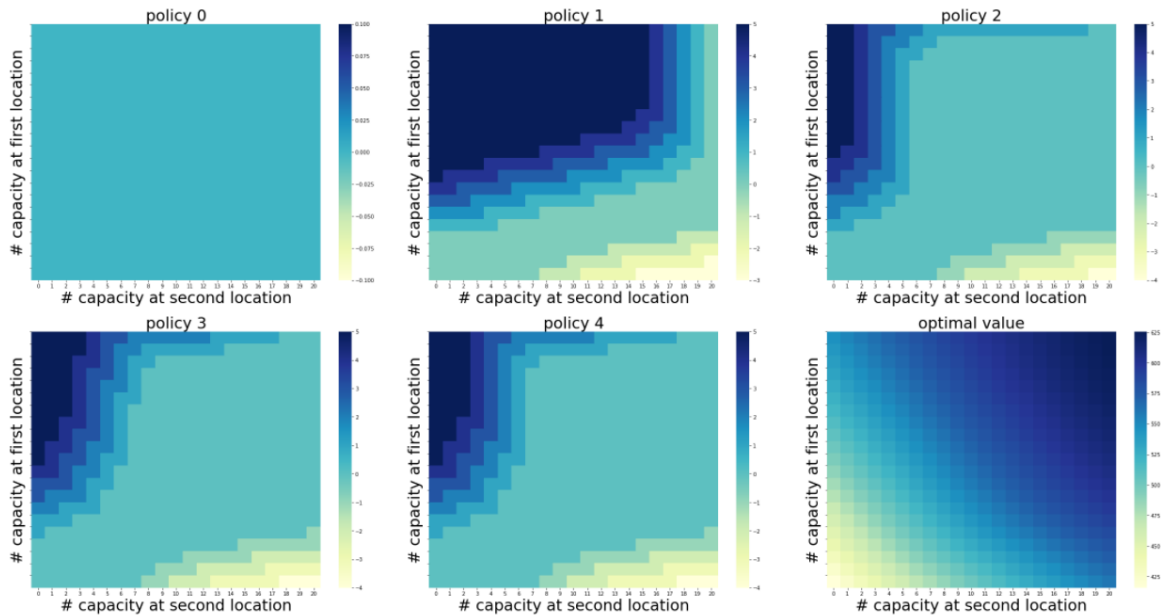


Figure 2: Representation of 'colorbar' for optimum policy

Changing the Purchase Cost from 2 to 6:

In this part we change **Purchase Cost** to 6 and then run the program to see how it can get a different result.

```
[ [ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -1]
[ 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 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 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]
[ 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 2 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 3 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 3 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 4 3 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 4 3 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 4 3 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 4 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 5 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 5 4 3 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 5 4 3 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 5 4 3 2 2 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[ 5 4 3 3 2 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

Figure 3: Representation of optimum Policy by changing Purchase Cost

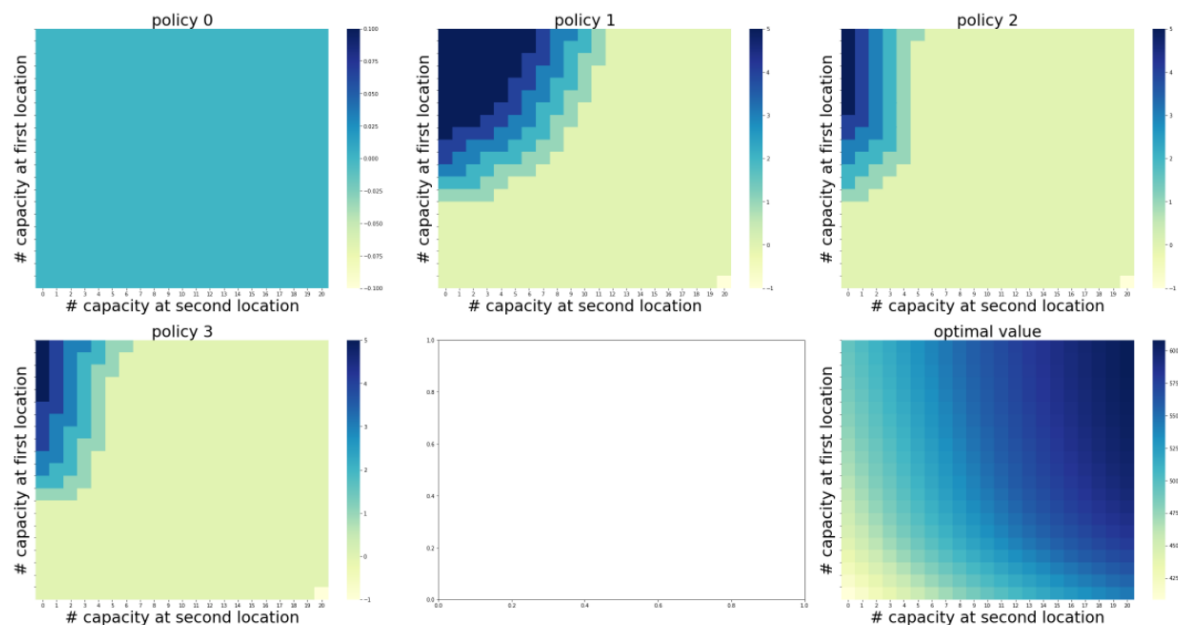


Figure 4: Representation of 'colorbar' for optimum Policy by changing Purchase Cost

Changing the Discount Factor from 0.9 to 1:

In this part we change the discount factor to one and then run the program to see how it can get a different result.

```
[
[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 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 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 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 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 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 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0]
[3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 0]
[4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 1 0]]
```

Figure 5: Representation of optimum Policy by changing Discount Factor

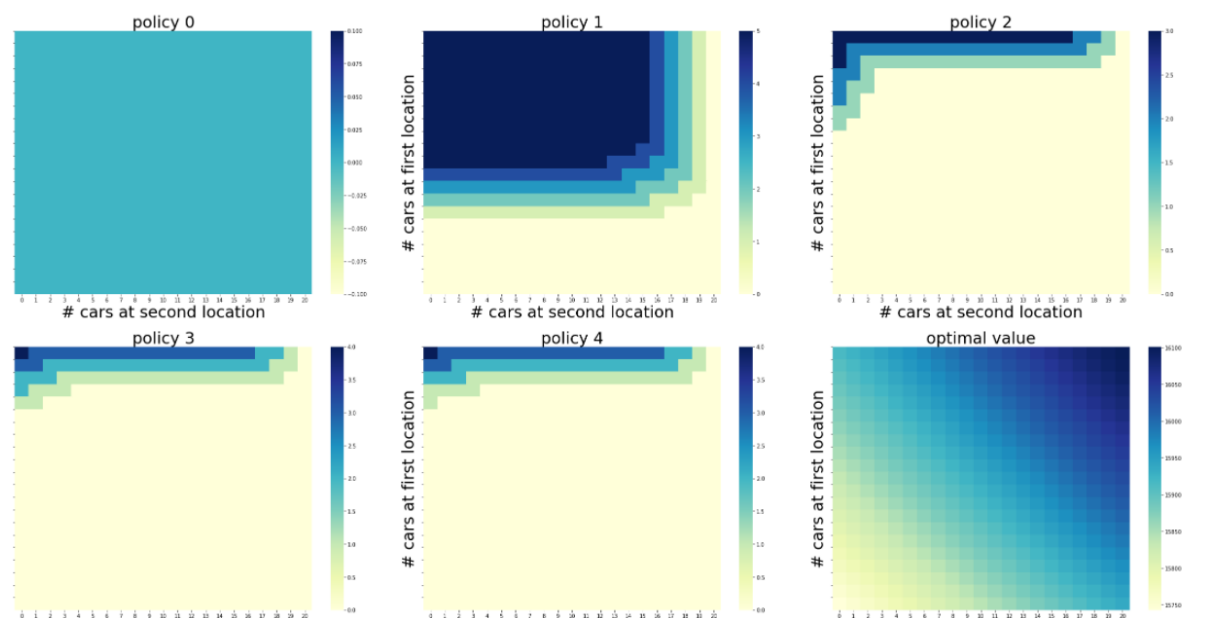


Figure 6: Representation of 'colorbar' optimum Policy by changing Discount Factor

As you can see the results for $\gamma=1$ is completely different with the previous process and as you can see the general forms of 'colorbar' has been destroyed and **also I have spent lots of time to run this part and approximately it took about 1 hour in my implementation it was so hard to debug in each effort but finally I made it true.**

Note that I have written a function which generate a subplot to show each step of iterations and correspond result as you can see in above figures.

Note that I have used a parametric approach in all of my functions which helped me to have a deeper approach and easier way to debug the code.

And finally this problem gives me a complete insight about model-based Reinforcement Learning and using bellmen equations in such a way that I used helped me to understand the theoretical approach behind algorithms like policy-iteration and also value-iteration.

3.2 MODEL-BASED REINFORCEMENT LEARNING(THEORETICAL)

In this part we intend to review the theoretical approach behind policy iteration algorithm using a 3x4 space.

We always go through desired path with probability of 0.6 and with a probability of 0.2 in two sides.

Our discounted factor in this case is equals to 0.2 and there is not any punishment for agent to do an action!

Note that in this problem we should repeat the steps for two separated iterations.

In each iteration we follow below steps:

- 1- initialize the policy for start moving around!
- 2- find the new value-functions using writing single equation for each state.
- 3- evaluating the policy using pi function under the policy (in other word just with checking the value of side state we can choose the best direction based-on a maximum approach for selecting the best action)
- 4- repeating the step 2 & 3 for under the new policy!

It's so amazing that we found just after two iterations we meet that many of direction located at the true one and it's so tricky and so nice in my opinion.

Furthermore, I want to talk about the simple note which I follow during my analyse, as we know when the table under any specific policy is made it's time to investigate whether it's the best one or not we can do that easily by checking the side value of states and decide what it should be instead of calculating all of pi function for all state which takes very time!

So I have calculated one example of policy improvement in each iteration because other have the same approach.

My theoretical analyse has been attached in the next page:

	(1)	(2)	(3)	(4)
(1)	→	←	↓	3
(2)	←	←	→	-2
(3)	→		↑	↑

First we choose an initial policy
we showed that using arrow notation
on the transition space.

now we write the value equation for all 8 states →

iteration 1

$$V(1,1) = 0 + \gamma 2 [\gamma 6 V(1,2) + \gamma 2 V(1,1) + \gamma 2 V(2,1)]$$

$$V(1,2) = 0 + \gamma 2 [\gamma 2 V(1,2) + \gamma 2 V(2,2) + \gamma 6 V(1,1)]$$

$$V(1,3) = 0 + \gamma 2 [\gamma 2 V(1,2) + \gamma 2 V(3,4) + \gamma 6 V(2,3)]$$

$$V(2,1) = 0 + \gamma 2 [\gamma 2 V(3,1) + \gamma 2 V(3,1) + \gamma 6 V(1,2)]$$

$$V(2,2) = 0 + \gamma 2 [\gamma 2 V(1,2) + \gamma 2 V(2,2) + \gamma 6 V(2,1)]$$

$$V(2,3) = 0 + \gamma 2 [\gamma 2 V(1,3) + \gamma 2 V(3,3) + \gamma 6 V(2,4)]$$

$$V(2,4) = -2$$

$$V(3,1) = 3$$

$$V(3,1) = 0 + \gamma 2 [\gamma 6 V(3,1) + \gamma 2 V(3,1) + \gamma 2 V(2,1)]$$

$$V(3,3) = 0 + \gamma 2 [\gamma 6 V(2,3) + \gamma 2 V(3,3) + \gamma 2 V(3,4)]$$

$$V(3,4) = 0 + \gamma 2 [\gamma 6 V(2,4) + \gamma 2 V(3,4) + \gamma 2 V(3,3)]$$

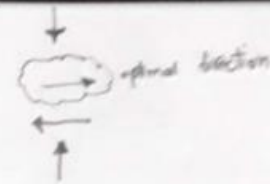
now we assume that the initial value functions are equals to zero so we have →
all $V(i,j)$ are equals to zero except →

$$\begin{cases} V(1,3) = \gamma 4 \times 3 = \boxed{\gamma 12} \\ V(2,3) = \gamma 12 \times -2 = \boxed{-\gamma 24} \\ V(2,4) = \boxed{-2} \\ V(3,1) = \boxed{3} \\ V(3,4) = \gamma 12 \times -2 = \boxed{-\gamma 24} \end{cases}$$

	(1)	(2)	(3)	(4)
(1)	0	0	$\gamma 12$	3
(2)	0	0	$-\gamma 24$	-2
(3)	0		0	$-\gamma 24$

now we must evaluate
the policy and if needed
update and improve the policy!

$$\pi_0(1,3) = \begin{cases} \gamma/2 [\gamma/6 V(2,3) + \gamma/2 V(1,4) + \gamma/2 V(1,2)] = 0.0912 \\ \gamma/2 [\gamma/6 \times 3 + \gamma/2 \times \gamma/12 + \gamma/2 \times (-\gamma/24)] = \boxed{0.3552} \text{ max} \\ \gamma/2 [\gamma/6 \times 0 + \gamma/2 \times \gamma/12 - \gamma/2 \times \gamma/24] = -0.0048 \\ \gamma/2 [\gamma/6 \times \gamma/12 + \gamma/2 \times 3 + \gamma/2 \times 0] = 0.1344 \end{cases}$$



and going through this way we also calculate $\pi_0(i,j)$ for every pair of (i,j)
 Note that we can easily find the optimal direction of each state by checking side
 V -values and such a way we can make below policy updating \rightarrow

iteration 2

	(1)	(2)	(3)	(4)
(1)	\rightarrow	\rightarrow	\rightarrow	3
(2)	\leftarrow	\leftarrow	\uparrow	-2
(3)	\uparrow		\uparrow	\leftarrow

iter ②

$$V(1,1) = 0 + \gamma/2 [\gamma/6 V(1,2) + \gamma/2 V(1,1) + \gamma/2 V(2,1)]$$

$$V(1,2) = 0 + \gamma/2 [\gamma/6 V(1,3) + \gamma/2 V(1,2) + \gamma/2 V(2,2)]$$

$$V(1,3) = 0 + \gamma/2 [\gamma/6 \times 3 + \gamma/2 V(1,2) + \gamma/2 V(2,3)]$$

$$V(2,1) = 0 + \gamma/2 [\gamma/6 V(2,1) + \gamma/2 V(3,1) + \gamma/2 V(1,1)]$$

$$V(2,2) = 0 + \gamma/2 [\gamma/6 V(2,1) + \gamma/2 V(2,2) + \gamma/2 V(1,2)]$$

$$V(2,3) = 0 + \gamma/2 [\gamma/6 \times \gamma/12 + \gamma/2 V(2,2) + \gamma/2 V(2,4)]$$

$$V(2,4) = -2$$

$$V(3,1) = 3$$

$$V(3,3) = 0 + \gamma/2 [\gamma/6 V(2,3) + \gamma/2 V(3,3) + \gamma/2 V(3,4)]$$

$$V(3,4) = 0 + \gamma/2 [\gamma/6 V(3,3) + \gamma/2 V(3,4) + \gamma/2 \times -2]$$

now according to previous value for V -values we update the new values all = 0 except:

$$\Rightarrow \begin{cases} V(1,2) = \gamma/2 \times \gamma/12 = \boxed{0.044} \\ V(1,3) = \gamma/2 [1.8 + \gamma/24 - \gamma/2 \times \gamma/24] = \boxed{0.2688} \\ V(2,3) = \gamma/2 [\gamma/6 \times \gamma/12 - \gamma/2 \times 2] = \boxed{-0.0656} \end{cases}$$

$$V(2,4) = \boxed{-2}$$

$$V(3,4) = \boxed{3}$$

$$V(3,3) = \gamma/2 [\gamma/6 \times -\gamma/24 + \gamma/2 \times -\gamma/24] = \boxed{-0.0384}$$

$$V(3,4) = \gamma/2 \times [-\gamma/24 \times \gamma/2 - 4] = \boxed{-0.896}$$

now according to the tips which discussed before we can check the table value instead of calculating policy π_i for each state!

so for reducing the calculations we do that:

	(1)	(2)	(3)	(4)
(1)	0	0.144	0.2688	3
(2)	0	0	-0.656	-2
(3)	0		-0.384	-0.896

so according to above table we have the policy updating just like below \rightarrow

	(1)	(2)	(3)	(4)
(1)	\rightarrow	\rightarrow	\rightarrow	3
(2)	\leftarrow	\uparrow	\uparrow	-2
(3)	\uparrow		\leftarrow	\leftarrow

4 QUESTION #3

4.1 UN-MODEL-BASED REINFORCEMENT LEARNING(Q-LEARNING)

In this part we intend to implement Q-Learning model through a simple target-tracker Game in python environment.

Note that the given file includes all of state rewards such as -13 reward belongs to the deeper area and also -3 reward belongs to walls!

One the important function in this case is. step, it gives us the four valuable output and lets us to know what is the next state number and also what is the reward which is belongs to specific transition and I think it's really nice and tricky to deal with such environment class in python.

Now let's have a deeper look on Q-learning method which I have used in this project:

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward.

When q-learning is performed we create what's called a q-table or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our q-values after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

The next step is simply for the agent to interact with the environment and make updates to the state action pairs in our q-table $Q[\text{state}, \text{action}]$.

Taking Action: Explore or Exploit

An agent interacts with the environment in one of two ways:

1)The first is to use the q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as exploiting since we use the information we have available to us to make a decision.

2) The second way to take action is to act randomly. This is called exploring. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. You can balance exploration/exploitation using epsilon (ϵ) and setting the value of how often you want to explore vs exploit.

Updating Q matrix:

The updates occur after each step or action and ends when an episode is done. Done in this case means reaching some terminal point by the agent. A terminal state for our case is getting target. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal q-values or q-star (Q^*).

Q-values are initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

$$Q(state, action) \leftarrow (1 - \alpha)Q(state, action) + \alpha \left(reward + \gamma \max_a Q(next\ state, all\ actions) \right)$$

Where:

α (alpha) is the learning rate ($0 < \alpha \leq 1$) - Just like in supervised learning settings, α is the extent to which our Q-values are being updated in every iteration.

γ (gamma) is the discount factor ($0 \leq \gamma \leq 1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas, a discount factor of **0** makes our agent consider only immediate reward, hence making it greedy!

Hyper-parameters and optimizations:

he values of 'alpha', 'gamma', and 'epsilon' were mostly based on intuition and some "hit and trial", but there are better ways to come up with good values!

α : (the learning rate) should decrease as you continue to gain a larger and larger knowledge base.

γ : as we get closer and closer to the deadline, your preference for near-term reward should increase, as you won't be around long enough to get the long-term reward, which means our gamma should decrease.

ϵ : as we develop our strategy, we have less need of exploration and more exploitation to get more utility from our policy, so as trials increase, epsilon should decrease.

now I want to discussed the functions which implemented in my code so please refer to the next page and have a look:

First of all we should reset the model and calling the GridworldEnv() just like what I did :

Reseting Model

```
env=GridworldEnv()
env.reset()
env._render()
```



I have declared the class which called env in whole of my code!

Now we have reached to the most important part of implementation which called training the model:

In this regard I have written a function called training-model() which gives all of necessary values such as Gamma, Learning rate, exploration-probability and also number of epoch!

```
def training_model(gamma, Learning_rate, exploration_probability, episode):
    Number_of_states=10*10*1
    Number_of_action=4
    Q_Matrix = loadtxt('Q_func_init.csv', delimiter=',')
    path_length_episode = []
    rewards_per_episode = []
    for i in range(episode):
        state = env.reset()
        move=0
        reward=0
        done = False
        total_episode_reward = 0
        while not done:
            if random.uniform(0, 1) < exploration_probability:
                action = env.action_space.sample()
            else:
                action = np.argmax(Q_Matrix[state])

            next_state, reward, done, info = env.step(action)
            if reward == -10 :
                next_state=44
            previous_value = Q_Matrix[state, action]
            next_max = np.max(Q_Matrix[next_state])
            #reward=reward-1
            new_value = (1 - Learning_rate) * previous_value + Learning_rate * (reward + gamma * next_max)
            Q_Matrix[state, action] = new_value
            total_episode_reward = total_episode_reward + reward
            state = next_state
            move += 1
        rewards_per_episode.append(total_episode_reward)

        if i % 100 == 0:
            clear_output(wait=True)
            print(f"Episode: {i}")
    print("Training succesfully finished.\n")
    savetxt('Q_func.csv', Q_Matrix, delimiter=',')

    return path_length_episode, rewards_per_episode, Q_Matrix
```

Some tips about my codes:

As we know our state space is $10*10*1=100$ because we have a $10*10$ grid as of our environment and also we have a target state which we suppose to track that!

And also we have four actions {Up,Down,Right,Left}.

Furthermore, we know that our Q-matrix has a 10*4 dimension as I pointed before!

So it's time to initialise the Q-matrix as we know the algorithm convergence is definitely so I decide to choose values from a `np.random.rand()` for the first time and then store their values in a .csv file in same directory and going through this way all times I easily read the value from a .csv file which is named `Q_Matrix_init.csv` !

You may be curious about this work but it's all about the value of random np array in every single run tend to change and so I have decided to store the initial value in directory and always read from .csv file!

And also I found in a try and error approach that randomly initializing is so tricky and has a good effect on performance and convergence speed so I decided to go through this way!

Implement returning to initial location when face with a deeper area!

In this regard I have used from the output of. step function such a way I each time check the value of reward and if it tends to be '-13' so I detected that it's the danger area because I haven't any experience in case of swimming!!

And after detecting I replace the value of state with single number of 44 as we know that's the initial location of our Subjective!

As I have pointed at the first part of my report I have used the mentioned equations to update my matrix and gain the new value for the cases!

Honestly the algorithm is so straight forward and so easy but as of the question description I am obligated to provide detailed information about each part of my code!

As I mentioned before in each of while looping I decide to choose one of exploration or exploitation, in this regard I have used from the parameter which is exploration-probability to decide how much it's likely to do exploration or maybe exploitation, as we know it's a better approach to decrease this value during iterations because when we are growing in our knowledge about the path we tend to use much more from our brain which is the q-matrix in correspond with our problem but cause of the further analyses in bonus part I decided to don't dealing with that!

There is also some another considerable notes which I want to mentioned in this part, if you care to my code you may see that I use a single code which always decrease the reward which minus one!

And also you may curious about this syntax, it's just for adding punishment for walking!

In other word, when I want to work on convergence speed I decided to add walking punishment so in this case our subjective should find their path as soon as it can and it's so common and popular in RL problems to add walking punishment to get a more efficient approaches.

And also in this part of code I calculate the rewards per episode to have a deeper insight on what is going on!

And at the final speech I have written a test cell to investigate the performance of the model.

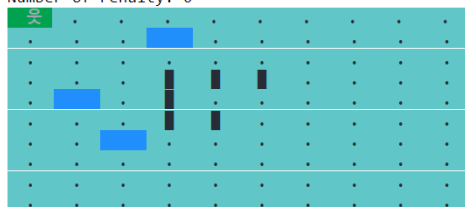
Testing Model After Q-Learning

```
episodes = 5
for _ in range(episodes):
    state = env.reset()
    move=0
    penalties=0
    reward = 0
    done = False
    while not done:
        action = np.argmax(Q_Matrix[state])
        state, reward, done, info = env.step(action)
        if reward == -10:
            penalties += 1
        if reward == -3:
            penalties += 1
        move += 1
    env._render()
    print(f"Results after {episodes} episodes:")
    print(f"Path Length: {move}")
    print(f"Number of Penalty: {penalties}")
```

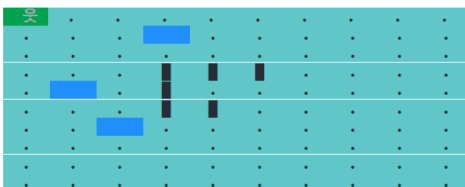
And now let's see the output:



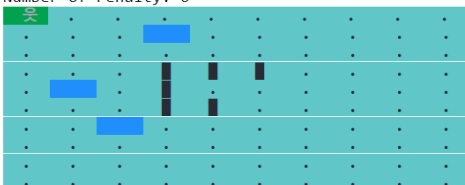
Results after 5 episodes:
Path Length: 12
Number of Penalty: 0



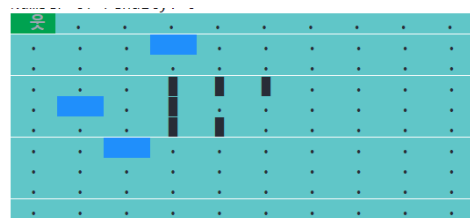
Results after 5 episodes:
Path Length: 12
Number of Penalty: 0



Results after 5 episodes:
Path Length: 12
Number of Penalty: 0



Results after 5 episodes:
Path Length: 12
Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

Number of Penalty: 0

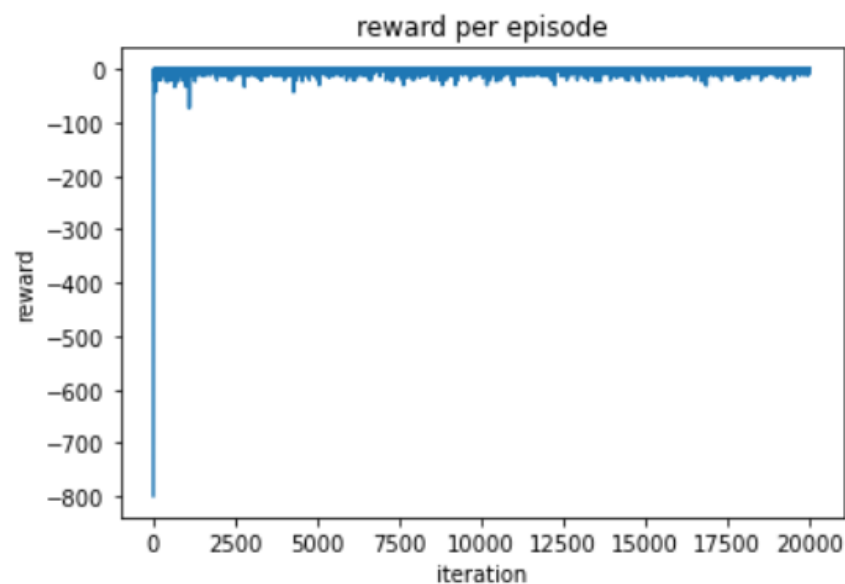
As you can see in all test cases we meet the convergence with the short possible path for five single test and so it shows that the training is in a right approach!

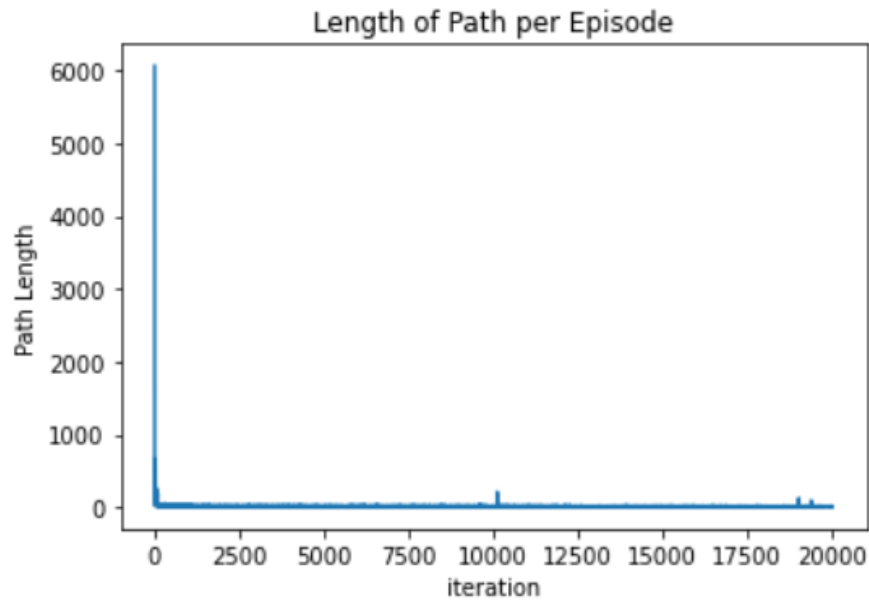
Please note that in I choose the epoch 20000 to meet the shortest path and also meet the target tracker goal!

And also I supposed that the convergence in this problem is about reaching target out and also getting the short path because in some cases I found that we may get the target but not in a shortest path, so it's so important for us to learn the model to track the target in an optimum way!

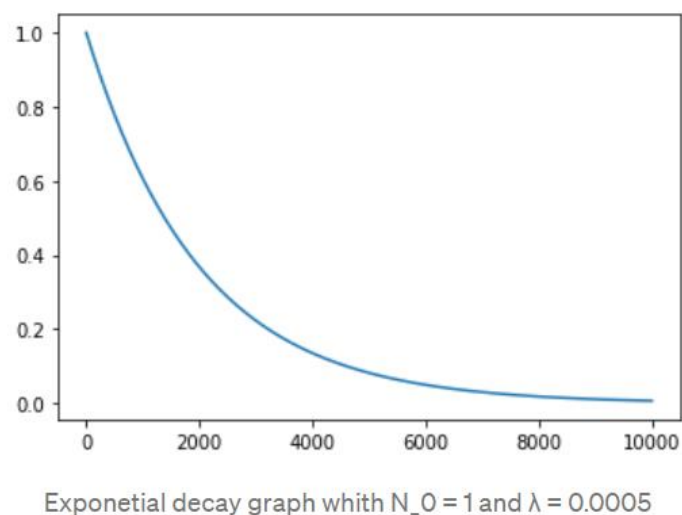
And as a considerable note it's much enough to train model for 20000 episodes to meet convergence as you can see the results in my code and also in my attached screenshots!

And the final step I want to attach the result of reward for each episode and also path length for each episode:



**Algorithm parameters:****Exploration-probability:**

In my implementation first I used the decay mode for exploration-probability as of an important factor to reaching out, because we expect when the epochs increase for enough number we learn the model in a good way and like to decrease the probability of exploration and just use the Q-table as of our brain although in large amount of epochs also we need to exploration but it should be choosing very small and in a dominant way we are supposed to go through our brain!



Gamma & Learning rate:

In two other parameters in this problem we don't need any sensitivity because according to my investigations the learning rate and also the Gamma should be the default value and we can meet the convergence so efficiently using constant mode for learning rate and also Gamma

So I have decided to choose $\text{Gamma}=0.99$ and $\text{learning rate}=0.1$

And these values are obtained in a try and error approaches! and as you can see can immediately can meet the convergence!

Bonus part:

In this part we are supposed to add some reward in such a way that we meet convergence with less epoch!

Note that as I mentioned previously I define the convergence in this problem getting the target and the shortest path so in this notation we must get convergence using less epochs!

Further I point to some of the most efficient approaches which satisfy our case:

1) Adding Walking punishment:

I considered the '-1' punishment for every transition in our state space!

In this way the model is forced to learn the optimum pass as soon as possible to eliminate increasing the steps because in each transmission it takes a negative reward and I have implemented this trick in my code in train function the results can be conducted below:

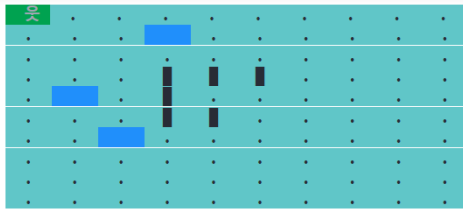
```
%%time
Learning_rate = 0.1
gamma = 0.99
exploration_probability = 0.1

path_length_episode, rewards_per_episode, Q_Matrix = training_model(gamma, Learning_rate, exploration_probability, 1000)

Episode: 900
Training succesfully finished.

Wall time: 882 ms
```

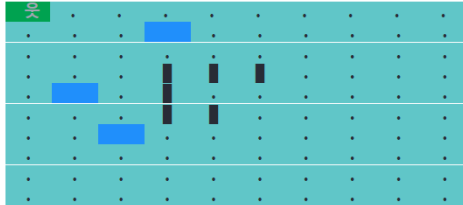
Not let's see the results:



Results after 5 episodes:

Path Length: 12

Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

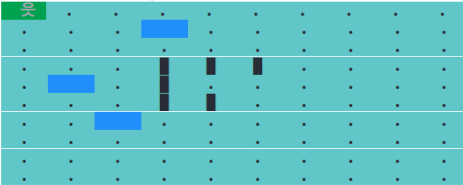
Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

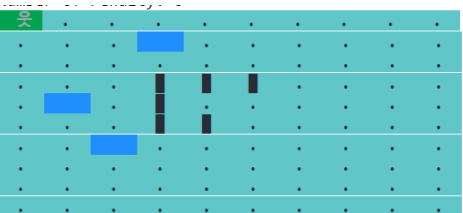
Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

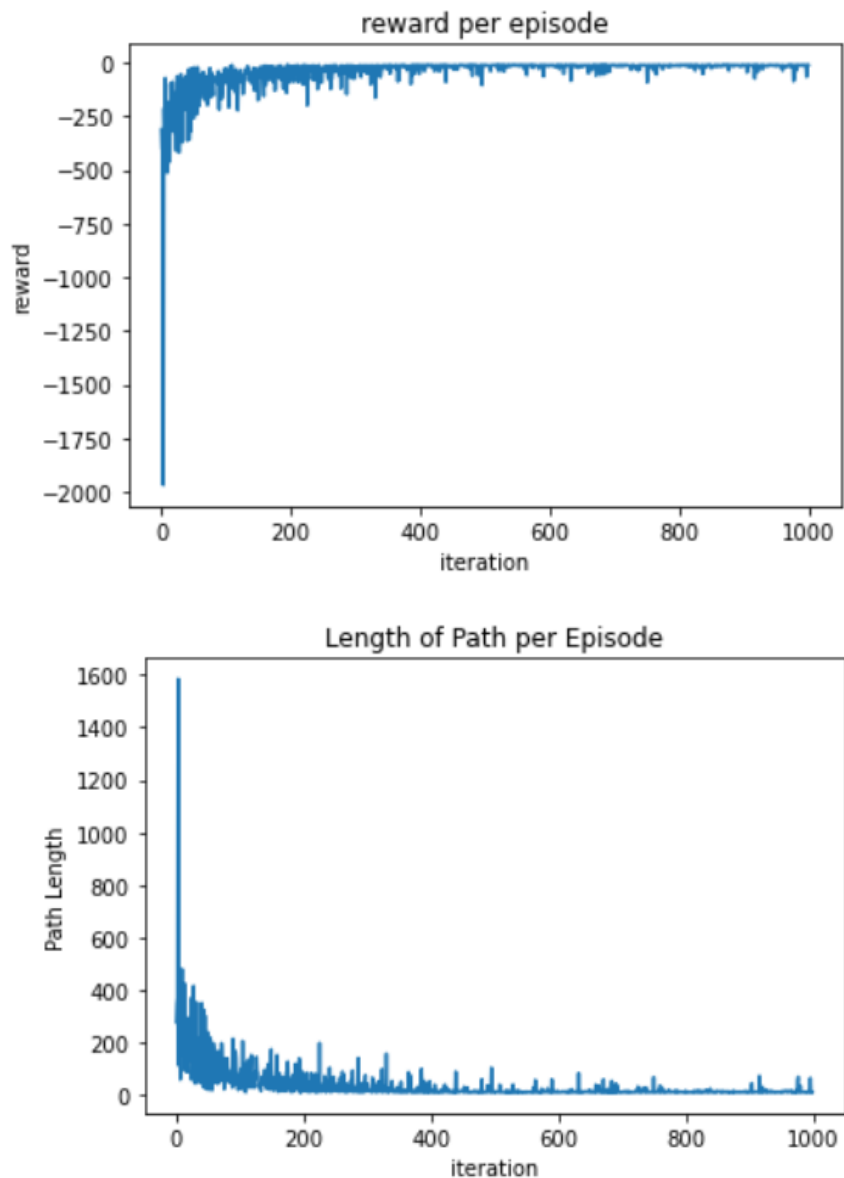
Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

Number of Penalty: 0



2) Adding Positive reward “+0.8” to Optimum Path!

I considered the ‘+0.8’ positive reward for every transition in our state space!

In this way I manually add some positive rewards to one of the optimum path in our map-set so the subjective should care more two these state because they take positive reward in comparison to those what never get rewards!

So in this way we can reach to the convergence in an easier and faster approaches.

States: $\{45, 46, 36, 26, 16, 6, 5, 4, 3, 2, 1\}$ should get +0.8 reward and note that in this way we must care that the reward must be less than 1 as of our target reward!

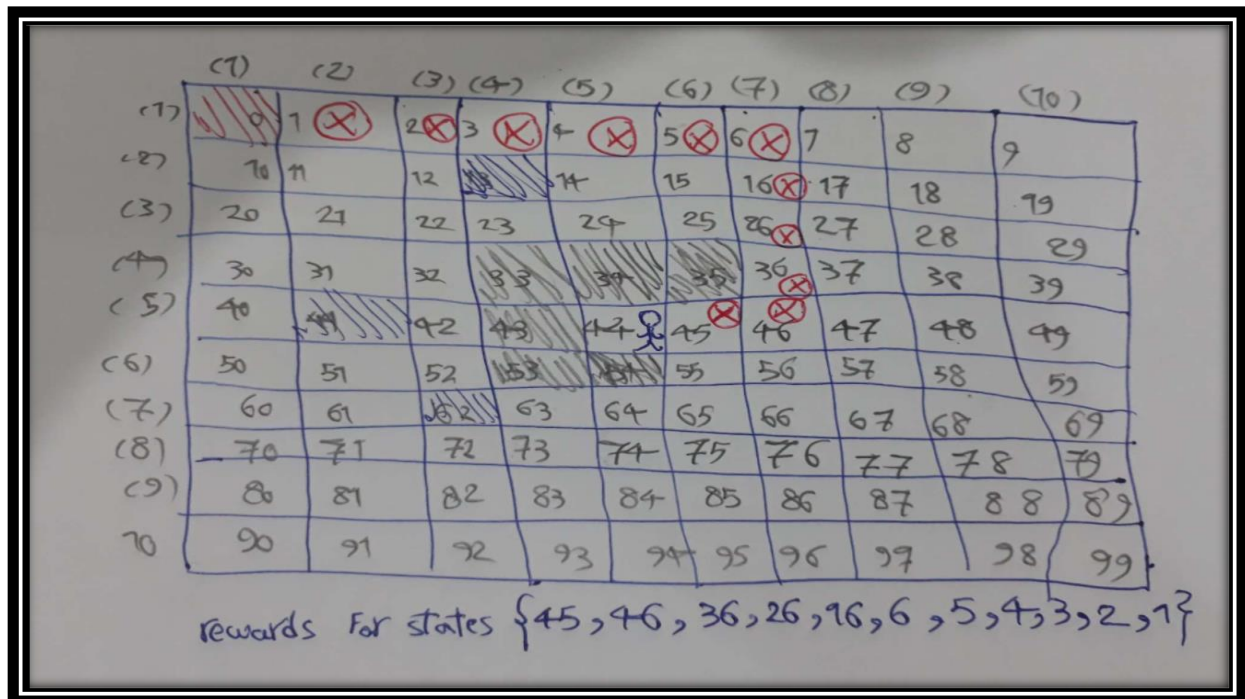


Figure 2: Representation of map-set

My implementation:

Second Idea

```
def training_model(gamma, learning_rate, exploration_probability, episode):
    Number_of_states = 10*10*1
    Number_of_action = 4
    Q_Matrix = loadtxt('Q_func_init.csv', delimiter=',')
    path_length_episode = []
    rewards_per_episode = []
    for i in range(episode):
        state = env.reset()
        move = 0
        reward = 0
        done = False
        total_episode_reward = 0
        while not done:
            if random.uniform(0, 1) < exploration_probability:
                action = env.action_space.sample()
            else:
                action = np.argmax(Q_Matrix[state])

            next_state, reward, done, info = env.step(action)

            if state == 45:
                reward = 0.8
            if state == 46:
                reward = 0.8
            if state == 36:
                reward = 0.8
            if state == 26:
                reward = 0.8
```

```

    if state == 26:
        reward=0.8
    if state == 16:
        reward=0.8
    if state == 6:
        reward=0.8
    if state == 5:
        reward=0.8
    if state == 4:
        reward=0.8
    if state == 3:
        reward=0.8
    if state == 1:
        reward=0.8
    if state == 1:
        reward=0.8

    if reward == -10 :
        next_state=44
    previous_value = Q_Matrix[state, action]
    next_max = np.max(Q_Matrix[next_state])
    reward=reward-1
    new_value = (1 - Learning_rate) * previous_value + Learning_rate * (reward + gamma * next_max)
    Q_Matrix[state, action] = new_value
    total_episode_reward = total_episode_reward + reward
    state = next_state
    move += 1
    rewards_per_episode.append(total_episode_reward)

```

```

    rewards_per_episode.append(total_episode_reward)
    path_length_episode.append(move)
    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")
    print("Training succesfully finished.\n")
    savetxt('Q_func.csv',Q_Matrix, delimiter=',')

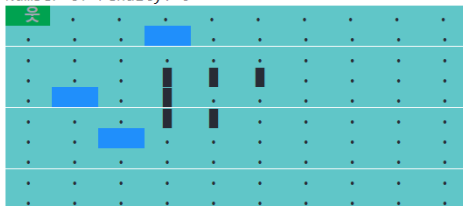
    return path_length_episode,rewards_per_episode,Q_Matrix

```

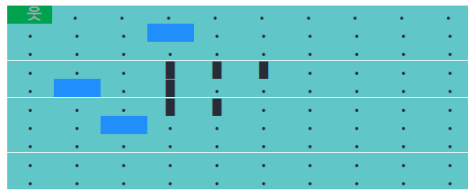
Not let's see the results:



Results after 5 episodes:
 Path Length: 12
 Number of Penalty: 0



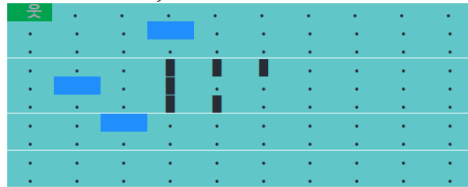
Results after 5 episodes:
 Path Length: 12
 Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

Number of Penalty: 0



Results after 5 episodes:

Path Length: 12

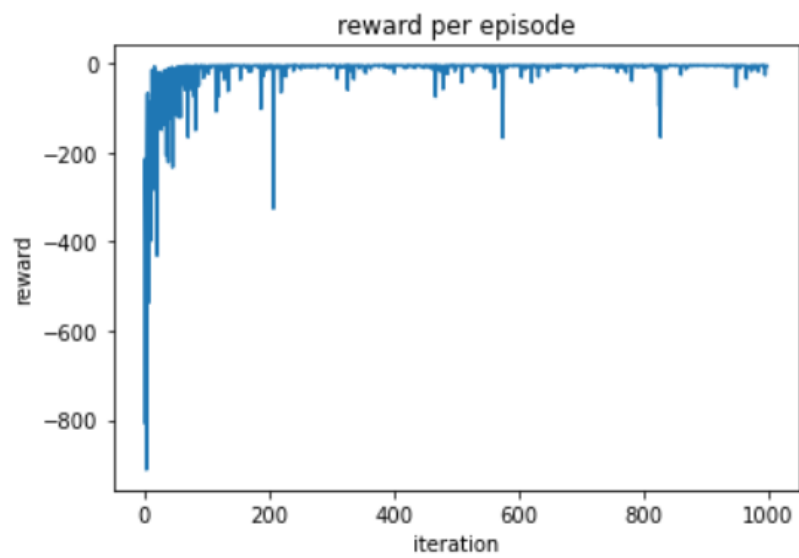
Number of Penalty: 0

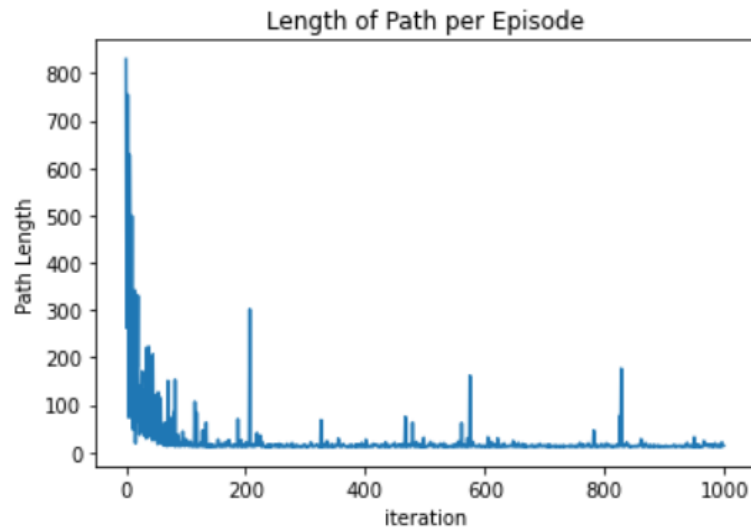


Results after 5 episodes:

Path Length: 12

Number of Penalty: 0





As you can see in both ideas we can decrease the epoch number from 20000 to 1000 which is so nice and so efficient in my opinion, and I think by these two approaches could satisfy the question target and That's it!

Here you can find all of my code implementation using Python if there is any issue with that please let me know to provide more detailed information. (be in touch [here!](#))

5 ACKNOWLEDGEMENT

I am really grateful for Mr MohammadReza Tavakoli (810197477) because in some part of this project we had a nice and effective discussion which highly helped us to have a better analyse and also have more adequate results! (Note that we only talked about ideas behind different problems.)

Afterwards, I am thankful to all of course teaching assistants: MahsaMassoud(mahsamassoud@gmail.com) NarjesNoorzad(njnoorzad@gmail.com) and and FatemeNoorzad(ati.noorzad@gmail.com) who designed this project with high quality.

6 REFERENCES

- [1] <https://medium.com/@rangavamsi5/na%C3%AFve-bayes-algorithm-implementation-from-scratch-in-python-7b2cc39268b9>
- [2] <https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming/>
- [3] <https://medium.com/@pesupavish/policy-iteration-easy-example-d3fd1eb98c6c>
- [4] <https://pythonprogramming.net/q-learning-algorithm-reinforcement-learning-python-tutorial/>
- [5] <https://www.geeksforgeeks.org/q-learning-in-python/>
- [6] <https://www.youtube.com/watch?v=RlugupBiC6w>
- [7] <https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56>
- [8] <https://realpython.com/python-conditional-statements/>
- [9] <https://towardsdatascience.com/policy-iteration-in-rl-an-illustration-6d58bdc87a7>