ANNAM SAIVARDHAN
200050008
200050008@iitb.ac.in

Programming Assignment 2
Foundations of Intelligent and Learning Agents (Autumn 2022)

2022-10-12

In this assignment, we will write code to compute an optimal policy for a given MDP using the algorithms that were discussed in class: Value Iteration, Howard's Policy Iteration, and Linear Programming. The first part of the assignment is to implement these algorithms. Input to these algorithms will be an MDP and the expected output is the optimal value function, along with an optimal policy. We also have to add an optional command line argument for the policy, which evaluates the value function for a given policy instead of finding the optimal policy, and returns the action and value function for each state in the same format.

MDP solvers have a variety of applications. As the second part of this assignment, we will use your solver to find an optimal policy for a batter chasing a target during the last wicket in a game of cricket.

1 MDP Planning

1.1 Problem Statement

Given an MDP, your program must compute the optimal value function V and an optimal policy Π by applying the algorithm that is specified through the command line. Create a python file called planner.py which accepts the following command-line arguments.

You are not expected to code up a solver for LP; rather, you can use available solvers as black-boxes. Your effort will be in providing the LP solver the appropriate input based on the MDP, and interpreting its output appropriately. Use the formulation presented in class. We recommend you to use the Python library PuLP

It is certain that you will face some choices while implementing your algorithms, such as in tie-breaking, handling terminal states, and so on. You are free to resolve in any reasonable way; just make sure to note your approach in your report.

1.2 Analysis

1.2.1 Value Iteration Continuing

Value iteration is a method of computing an optimal MDP policy and its value.Value iteration starts at the ëndänd then works backward, refining an estimate of either Q or V. There is really no end, so it uses an arbitrary end point. We define an arbitary V and then recursively make it better. We stop the recursion when the improvement is less.Let us see the code snippet for it.

```
def value_iteration_cont(numStates,numActions,transition_matrix,reward_matrix,discount_factor,epsilon):
    # Initialize Value_Function arbitrarily
    V1 = np.zeros(numStates)
    V2 = np.zeros(numStates)
    # Initialize optimal_policy arbitrarily
    policy = np.zeros(numStates)
    # Initialize differences arbitrarily
    diff = np.zeros(numStates)
    # Repeat until convergence
    while True:
        # For each state s
        for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                # Compute the value of each action
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor * V1))
            # Select the best action
            best_action = np.argmax(action_values)
            # Update the policy
            policy[s] = best_action
            # Update the value function
            V2[s] = action_values[best_action]
            diff[s] = np.abs(V1[s] - action_values[best_action])
        # Check for convergence
        if np.all(diff < epsilon ):
            break
        for s in range(numStates):
            # Update the value function
            V1[s] = V2[s]
    return V1, policy
```

Value Iteration Continuing Code snippet

1.2.2 Policy Iteration Continuing

Policy Iteration Continuing is an algorithm to find an optimal MDP policy. Here at every step we update the existing policy using the Q values. The important part is that we never get the same policy twice. We always move in one direction. We stop when there is no scope for improvement. The policy at which we stop is the optimal policy.Let us see the code snippet for it.

```
#######FUNCTION TO SOLVE THE MDP USING HOWARD POLICY ITERATION FOR CONTINUOUS MDP######
def howard_policy_iteration_cont(numStates,numActions,transition_matrix,reward_matrix,discount_factor,epsilon):
    # Initialize Value_Function arbitrarily
    V = np.zeros(numStates)
    # Initialize optimal_policy arbitrarily
    policy = np.zeros(numStates)
    #Variable to store whether improvement is possible or not
    improve=True
    while (improve):
        improve = False
        # For each state s
        #Compute the policy evaluation for current policy
        V = policy_evaluation(policy,numStates,numActions,transition_matrix,reward_matrix,discount_factor)
        for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor * V))
            # Select the best action
            best_action = np.argmax(action_values)
            b = best_action
            # Check for improvement
            if policy[s]!=best_action and action_values[b]>action_values[(int)(policy[s].item())]:
                improve=True
            # Update the policy
            policy[s] = best_action
    return V, policy
```

Howard Policy Iteration Continuing Code snippet

1.2.3 Linear Programming Continuing

Linear Programming is a standard method to find an optimal solution to an MPD problem. Here we consider the MDP policy evaluation values as variables and write some inequalities and by using our knowledge on vector spaces we write a minimization problem on our variables. Now our Linear solver uses this constraints and provide a solution to us. If there are multiple optimal solutions, our solver gives one solution among them. Here we are using PuLP python library to do Linear Programming.Let us see the code snippet for it.

```
def linear_programming_cont(numStates,numActions,transition_matrix,reward_matrix,discount_factor):
    #Initializing the LP problem
    problem = pulp.LpProblem("MDP", pulp.LpMinimize)
    #Initializing the variables
    decision_variables = []
    for s in range(numStates):
        variable = str('v'+str(s))
        decision_variables.append(pulp.LpVariable(variable, lowBound=0))
    #Initializing the objective function
    problem+= (np.sum(decision_variables))
    #Initializing the constraints
    for s in range(numStates):
        for a in range(numActions):
            problem+= (decision_variables[s]>= np.sum(((transition_matrix[s,a,:]*discount_factor) * ((reward_matrix[s,a,:]/discount_factor) +  decision_variables))))
    #Solving the LP problem
    problem.solve(pulp.PULP_CBC_CMD(msg=False))
    #Extracting the V values
    V = np.zeros(numStates)
    for i in range(numStates):
        V[i]=np.float64(decision_variables[i].value())
    #Extracting the policy
    policy = np.zeros(numStates)
    for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor * V))
            best_action = np.argmax(action_values)
            # Update the policy
            policy[s] = best_action
    return V, policy
```

Linear Programming Continuing Code snippet

1.2.4 Value Iteration Episodic

Value Iteration Episodic is method to find optimal solution of an MDP where the number of steps is finite , i.e the process stops after reaching the end states. End states are like the final destination. Here we use an algorithm similar to the one we used in Value Iteration Continuing.The policy we get finally is the optimal policy.Let us see the code snippet for it.

```python
def value_iteration_episodic(numStates,numActions,transition_matrix,reward_matrix,discount_factor,epsilon):
    V = np.zeros(numStates)
    V2 = np.zeros(numStates)
    # Initialize optimal policy arbitrarily
    policy = np.zeros(numStates)
    # Initialize differences arbitrarily
    diff = np.zeros(numStates)
    # Repeat until convergence
    c=0
    while True:
        # For each state s
        for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor*V))
            # Select the best action
            best_action = np.argmax(action_values)
            # Update the policy
            policy[s] = best_action
            # Update the value function
            V2[s] = action_values[best_action]
            diff[s] = np.abs(V[s] - action_values[best_action])

        # Check for convergence
        if np.all(diff < epsilon ):
            break
        #print(diff)
        for s in range(numStates):
            V[s] = V2[s]
    return V, policy
```

Value Iteration Episodic Code snippet

1.2.5 Policy Iteration Episodic

Policy Iteration Episodic is an algorithm to find an optimal MDP policy. Here at every step we update the existing policy using the Q values. The important part is that we never get the same policy twice. We always move in one direction. We stop when there is no scope for improvement.The difference between Episodic and Continuing for this method is that, in Episodic we reach end states and stop there whereas in Continuing we go on forever. The policy at which we stop is the optimal policy.Let us see the code snippet for it.

```python
######FUNCTION TO SOLVE THE MDP USING HOWARD POLICY ITERATION FOR EPISODIC MDP######
def howard_policy_iteration_episodic(numStates,numActions,transition_matrix,reward_matrix,discount_factor,epsilon):
    # Initialize Value Function arbitrarily
    V = np.zeros(numStates)
    # Initialize optimal policy arbitrarily
    policy = np.zeros(numStates)
    #Variable to store whether improvement is possible or not
    improve=True
    while (improve):
        improve = False
        # For each state s
        #Compute the policy evaluation for current policy
        V = policy_evaluation(policy,numStates,numActions,transition_matrix,reward_matrix,discount_factor)
        for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor * V))
            # Select the best action
            best_action = np.argmax(action_values)
            b = best_action
            # Check for improvement
            if policy[s]!=best_action and action_values[b]>action_values[(int)(policy[s].item())]:
                improve=True
            # Update the policy
            policy[s] = best_action
    return V, policy
```

Policy Iteration Episodic Code snippet

1.2.6 Linear Programming Episodic

Linear Programming is a standard method to find an optimal solution to an MPD problem. Here we consider the MDP policy evaluation values as variables and write some inequalities and by using our knowledge on vector spaces we write a minimization problem on our variables. Now our Linear solver uses this constraints and provide a solution to us. If there are multiple optimal solutions, our solver gives one solution among them.

Here we are using PuLP python library to do Linear Programming.Let us see the code snippet for it.

```python
def linear_programming_episodic(numStates,numActions,transition_matrix,reward_matrix,discount_factor):
    #Initializing the LP problem
    problem = pulp.LpProblem("MDP", pulp.LpMinimize)
    #Initializing the decision variables
    decision_variables = []
    for s in range(numStates):
        variable = str('v'+str(s))
        decision_variables.append(pulp.LpVariable(variable, lowBound=0))
    #Initializing the objective function
    problem+= (np.sum(decision_variables))
    #Initializing the constraints
    for s in range(numStates):
        for a in range(numActions):
            problem+= (decision_variables[s]>= np.sum(((transition_matrix[s,a,:]*discount_factor) * ((reward_matrix[s,a,:]/
            discount_factor) +  decision_variables))))
    #Solving the LP problem
    problem.solve(pulp.PULP_CBC_CMD(msg=False))
    #Extracting the V values
    V = np.zeros(numStates)
    for i in range(numStates):
        V[i]=np.float64(decision_variables[i].value())
    #Extracting the policy
    policy = np.zeros(numStates)
    for s in range(numStates):
            # Compute the value of each action
            action_values = np.zeros(numActions)
            for a in range(numActions):
                action_values[a] = np.sum(transition_matrix[s,a,:] * (reward_matrix[s,a,:] + discount_factor * V))
            best_action = np.argmax(action_values)
            # Update the policy
            policy[s] = best_action
    return V, policy
```

Linear Episodic Episodic Code snippet

2 Cricket: The last wicket

2.1 Prablem Statement

The following problem is based on the game of Cricket. In Cricket, there are two teams, each with 11 players- the batting and the bowling team, which play on a pitch (between 2 wickets) located at the centre of the ground. The batting side scores runs by striking the ball bowled at the wicket with the bat and then running between the wickets, while the bowling and fielding side tries to prevent this (by preventing the ball from leaving the field, and getting the ball to either wicket) and dismiss each batter (so they are "out"). At any instance there are 2 players (striker and non-striker) from the batting team, and 11 players from the bowling team (1 bowler, 10 fielders) on the field. After every 6 balls (also called over) the striker and non-striker swap their positions: that is, the non-striker becomes a striker and vice-versa. A striking batter can either get dismissed (or out) or can score one of 0, 1, 2, 3, 4, 6 runs. To score 0, 1, 2, or 3 runs the batters need to swap their positions respective amount of times in the same ball. A striking batter can hit a 4 or 6 if the ball crosses the boundary (6 incase the ball crosses aerially).

Consider two batters A and B. In this task we aim to find the optimal policy for batter A, assuming we have no control over the actions of batter B: that is, batter A is the agent and batter B along with rest of game dynamics is part of the environment. A is a middle-order batter at the last wicket, along with a tail-ender "B". Additionally, consider that B can only either get out or score 0/1 runs. As part of the environment, we can fix a parameter "q" indicating the degree of weakness of B (a detailed description is given in the next section) The batting team still has to get T runs ($T \leq 30$) in O balls ($O \leq 15$). We can formulate this problem as an MDP. The set of states is encoded in the form bbrr, where bb and rr are 2-digit numbers representing the number of balls left, and the number of runs to score to reach the target. Single digit numbers have a 0 attached to the left to reach 2 digits (eg - 0701 for 1 run to score in 7 balls).

2.2 Solution

Here the problem is divided into 3 parts. We need to first encode and then feed the output of this encoder to planner.py and then the output of planner.py to decoder.py

2.2.1 encoder.py

Encoder.py python code takes the list of states provided to it and encodes it to MDP problem. Let me explain how I encoded it into MDP.

NumStates: Number of states in our MDP problem. In my MDP encoding numStates = 2states+2 where states are number of states provided to us in the input. Here the first numStates/2 -1 states correspond to the case where batter A is batting and the next numStates/2-1 to the batter B. The last two states correspond to win and lose. Game ends when someone reach those states.
NumActions: There are five actions possible. Batsman can try to hit 0,1,2,4,6 and these corresponds to actions 0,1,2,3,4 .
Transitions: Using the probabilities of score for a single ball for different actions given in the input file I calculated the probabilities of Transitions between states.
End states: Here the end states are the winning and losing states which are last 2 states.
MDPtype: As we have end states we solve this in episodic style.
Let us see the code snippets of it.

```
    #OPENING THE STATES FILE
    states = open(args.states,'r')
    lines = states.readlines()
    num_states = len(lines)
    #TRANSITION PROBABILITIES MATRIX
    transition_matrix = np.zeros((2*num_states+2,2*num_states+2,5))
    #REWARD MATRIX
    reward_matrix = np.zeros((2*num_states+2,2*num_states+2))
    reward_matrix[:,2*num_states:2*num_states+1]=1
    #DICTIONARY TO STORE STATE NUMBERS TO ZERO INDEXING MAPPING
    dict = {}
    cnt=0
    for line in lines:
        arr = line.split()
        dict[int(arr[0])] = cnt
        cnt+=1
    #WIN KEY IS SET TO 11111
    dict[11111] = 2*num_states
    #LOSS KEY IS SET TO 99999
    dict[99999] = 2*num_states+1
```

Code Snippet To Open File And Read States From It And Initialise Matrices.

```
    for line in lines:
        arr = line.split()
        num = int(arr[0])
        rr  = int(num%100)
        bb  = int((num//100)%100)
        init_state = int(dict[num])
        if bb>1:
            #FOR PLAYER1 STATES
            transition_matrix[int(init_state),int(dict[99999]),:] = p1_actions[:,0]
            for i in range(6):
                j = 6 if i==5 else i
                if rr-j>0:
                    next_state_ = dict[num-100-j] if ((j%2==0 and bb%6!=1) or (j%2==1 and bb%6==1)) else num_st
                    transition_matrix[int(init_state),int(next_state_),:] += p1_actions[:,i+1]
                else:
                    next_state_=dict[11111]
                    transition_matrix[int(init_state),int(next_state_),:] += p1_actions[:,i+1]

            #FOR PLAYER2 STATES
            init_state2 = init_state + num_states
            # OUT
            transition_matrix[int(init_state2),dict[99999],:] += [p2,p2,p2,p2,p2]
            if bb%6!=1:
                # ZERO RUNS
                nextstat = dict[num-100]+num_states
                zz=(1-p2)/2
                transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz]
                nextstat = dict[num-101] if rr>1 else dict[11111]
                transition_matrix[int(init_state2),int(nextstat),:]+= [zz,zz,zz,zz,zz]
```

The above photo is the snippet of how we are assigning the transition probabilities based on ball remaining and runs remaining.

```
            if bb%6!=1:
                # ZERO RUNS
                nextstat = dict[num-100]+num_states
                zz=(1-p2)/2
                transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz]
                nextstat = dict[num-101] if rr>1 else dict[11111]
                transition_matrix[int(init_state2),int(nextstat),:]+= [zz,zz,zz,zz,zz]
            else:
                zz=(1-p2)/2
                #ZERO RUNS
                nextstat = dict[num-100]
                transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz]
                nextstat = dict[num-101]+num_states if rr>1 else dict[11111]
                transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz]
        else:
            #FOR PLAYER1 STATES
            transition_matrix[int(init_state),int(dict[99999]),:] += p1_actions[:,0]
            for i in range(6):
                j = 6 if i==5 else i
                next_state_ = dict[99999] if rr-j>0 else dict[11111]
                transition_matrix[int(init_state),int(next_state_),:] += p1_actions[:,i+1]
            #FOR PLAYER2 STATES
            #ZERO RUNS
            nextstat = dict[99999]
            init_state2 = init_state + num_states
            zz = (1+p2)/2
            transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz] #zero runs+ out probability
            zz=(1-p2)/2
            nextstat = dict[99999] if rr>1 else dict[11111]
            transition_matrix[int(init_state2),int(nextstat),:] += [zz,zz,zz,zz,zz]
```

Code Snippet To Assign Transition Values and Rewards.

2.2.2 Planner.py

: Planner.py is the same code we coded in Task1. We use the default solver in task 1 to solve our MDP.

2.2.3 Decoder.py

: Decoder.py takes the output from planner.py and then using this and states from states file it creates the optimal Policy for our problem.Let us see the code snippet for it.

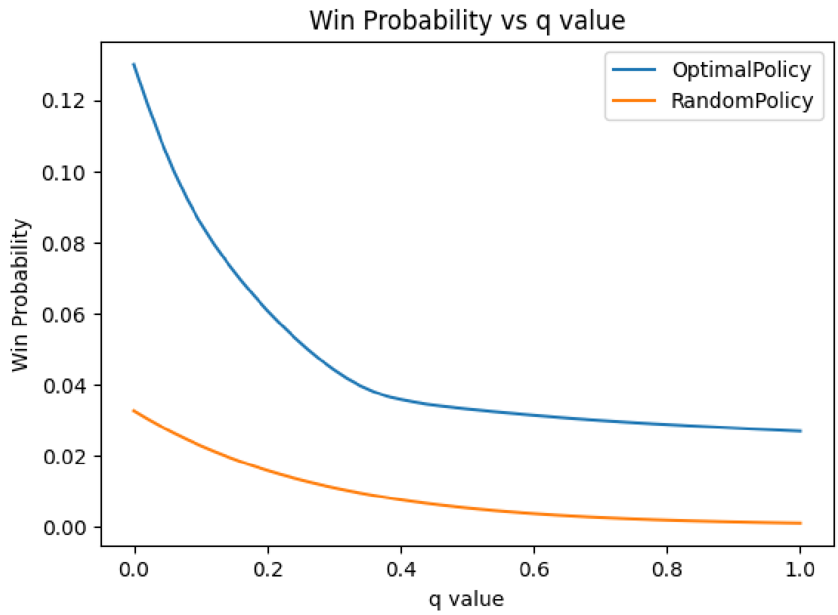
```

    #OPENING THE STATES FILE
    stat_file = open(args.states,'r')
    states = stat_file.readlines()
    stat_list = []
    for state in states:
        stat_list.append(state.split()[0])




    #OPENING THE VALUE AND POLICY FILE
    val_pol_file = open(args.value_policy,'r')
    val_pol = val_pol_file.readlines()
    cnt = 0
    for line in val_pol:
        arr = line.split()
        a = int(float(arr[1]))
        z = a if a<3 else int(a+1)
        z = 6 if z==5 else z
        print(str(stat_list[cnt])+' '+str(z)+' '+str(arr[0]))
        cnt+=1
        if cnt>=150:
            break
    stat_file.close()
    val_pol_file.close()

```

Decoder code snippet to Decode output from planney.py.

2.3 Graphs

2.3.1 Win Probability vs B's strength(q value)

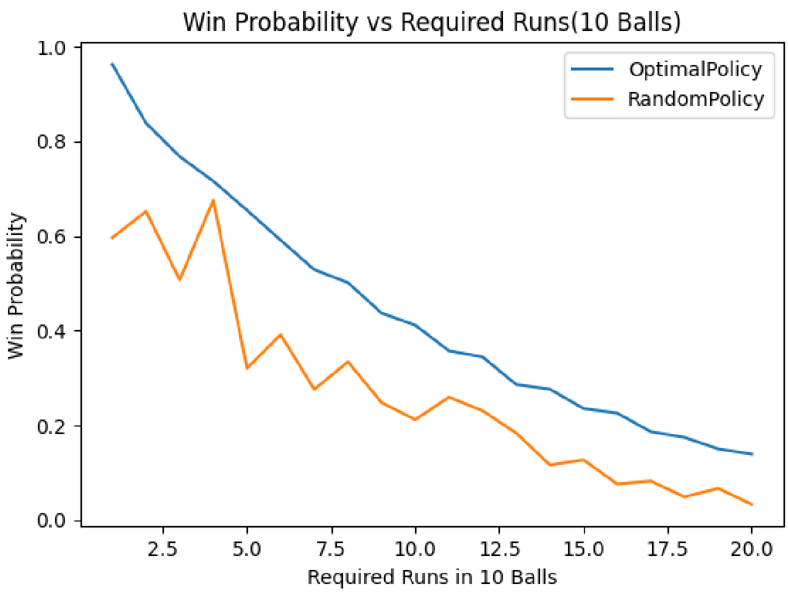
As you can see, the optimal policy probability is always more than the random policy as expected. And as the value of q increases win probability is decreasing, which is intuitively correct. As q increases the probability of B getting out increases so hence the probability of winning decreases.



Probability of Win vs B's Strength(q value).

2.3.2 Win Probability vs Runs Required(in 10 balls)

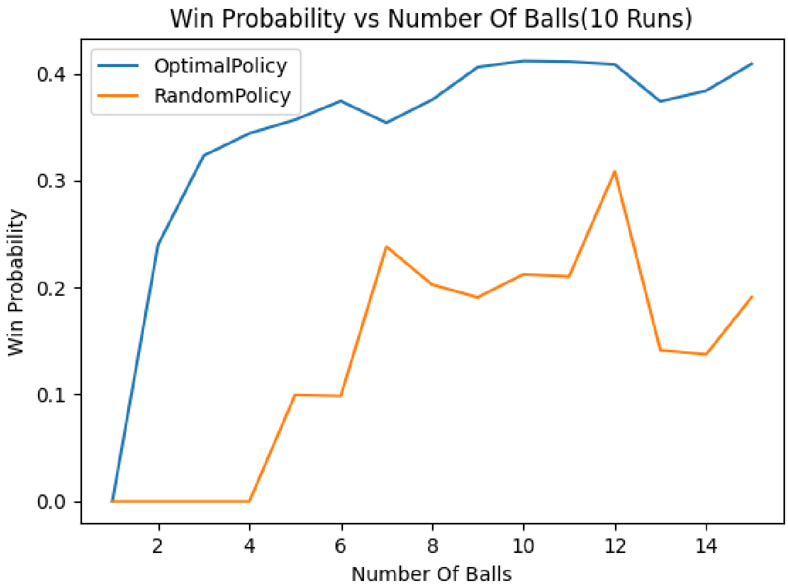
As you can see, similar to above, the optimal policy is performing better than random policy as expected. Here the optimal policy probability of winning is strictly decreasing as expected but when you see the random policy the decrease is not consistent. At some places as runs increases the probability of winning is increasing, this behaviour is the result of random policy. As the runs increases new states gets introduced. In random policy if the current policy is a bad one and after adding some more states the policy values for new states may reduce the bad effect of old policy and hence the probability is increasing sometimes.



Probability of Win vs Required Runs(10 Balls).

2.3.3 Win Probability vs Balls(10 Runs)

Similar to above graphs the performance of the optimal policy is better than the random policy. Both of them have zero probability of winning when balls are 1. But the probability of winning is zero until four balls for random policy whereas it is non zero for optimal one. And one important observation is that the probability of winning is changing drastically for random policy at balls=6 and balls=12. This is because of the over change of the batsman. And also this over change is responsible for decrease of probability of winning near ball=6 and balls=12 for optimal policy.



Probability of Win vs No Of Balls(10 Runs).