Missouri University of Science and Technology, Rolla

BAYESIAN MODELING OF HUMAN SEQUENTIAL DECISION-MAKING ON MULTI ARMED BANDIT PROBLEMS

SEMESTER PROJECT REPORT

Submitted On: 11th December 2018

CS 6202: Markov Decision process

Mounica Devaguptapu

Student ID: 12558500

E-Mail ID: md2rt@mst.edu

Department: Computer Science

Specialization: Human Decision Making

Expected Graduation Date: May 2020

Previous Degree: Bachelor's in Technology

Table of Contents

1 Executive Summary

Sequential decision-making in uncertain environments form an important class of problems in which agent must simultaneously learn about the environment while choosing among uncertain alternatives to gather reward. Human engage in sequential decision making in everyday life. The ability to perform correct decisions has high-impact in the quality of life. Multi-Armed Bandit Problem (MAB) offers a good opportunity to test human sequential decision-making given its simplicity and widely-known optimal solution through Gittins Index. As a part of this project, we will generate a multi-armed bandit environment where human (agent) will be making sequential decisions aiming to maximize the end reward. Also, we will implement different belief updating models that depicts the human behavior and find the best predictive performance among them.

Keywords: Sequential decision making, Multi-Armed Bandit problem, Gittins Index, Belief updating models

2 Introduction

2.1 Markov Decision Process

To understand Markov Decision Process, first we must first understand Markov Property.

Markov property states that,

"The future is independent of the past given the present"

Markov Decision Processes are a class of problems where an agent is supposed to decide the best action to select at every step following Markov property.

A Markov Decision Process (MDP) model contains:

- A set of possible world states S.
- A set of Models
- A set of possible actions A.
- A real valued reward
- A policy the solution of Markov Decision Process.

A *State* is a set of tokens that represent every state that the agent can be in.

An *Action* is the set of all possible options that the agent has at any given state.

A *Model* is something that mimics the behavior of the environment. It is way that decides on a course of action by considering all possible future situations before they are experienced.

A *Policy* defines learning agent's way of behaving at a given time.

A *Reward* defines the goal of the problem. At any given state, choosing an action produces a reward and the agent's sole objective is to maximize total reward.

2.2 Multi-Armed Bandit Problem

Multi-Armed Bandit Problem is a problem in which a fixed set of resources must be allocated between competing choices in a way that maximizes their expected gain, when each choice's properties are only partially known at the time of allocation and may become better understood as time passes or by allocating resources to the choice.

In MAB problems, selecting arm *i* at time *k* generates a reward x_k^i. Our aim is to construct an action selection policy that maximizes the expected total discounted future reward $x_0 + \gamma x_1 + \gamma^2 x_2 + \cdots \ldots$ where γ ($0 \leq \gamma \leq 1$) is the discounting factor that allows the infinite sum to converge.

In building a policy (defines which arm to pick at time t), if we select arm I at a time t steps in the future, we expect a reward $\gamma^t r(\pi_{k+t}) where\ r(\pi) = \mathbb{E}[x|\pi]$ is the expected reward of the state π and $\mathbb{E}$ is the expectation operator. Thus, MAB problems are special cases of Markov Decision Processes, and hence have an associated Bellman equation which can be solved via dynamic programming.

Gittins proposed a solution to MAB problems in the form of an index which is explained later sections.

2.3 Gittins Index

Gittins (Gittins, 1989) proved that the solution to MAB problems takes the form of an index for each arm, called *Gittins Index*, and that the optimal action at each decision time is to pull the arm with highest index.

Gittins Index is a virtue as only information from a particular arm's dynamics is required to compute that arm's index. The index is the ratio of the discounted expected reward if the arm is assumed to be pulled till the stopping time and the total discounted time.

Gittins Index for arm *'i'* is given by,

$$G_i = \mathbb{E}_{\pi^i}\left[\sum_{t=0}^{\tau-1}\gamma^t\, r(\pi_i^t)\right] \Big/ \mathbb{E}_{\pi^i}\left[\sum_{t=0}^{\tau-1}\gamma^t\right]$$

$where\ r(\Pi_i^t)\ is\ the\ reward\ obtained\ by\ choosing\ arm\ i$ at time t.

The value function of the Multi-Armed Bandit problem using Gittins Index is

$$v_i(\pi^i) = \sup_{\tau>0} \mathbb{E}_{\pi^i}\left[\sum_{t=0}^{\tau-1}\gamma^t\, r(\pi_i^t)\right] \Big/ \mathbb{E}_{\Pi^i}\left[\sum_{t=0}^{\tau-1}\gamma^t\right]$$

Another way of computing Gittins index of an arm is using **Calibration Method**. This method calibrates an arm by comparing it with a standard bandit process, which has one state and a constant reward λ. This method works by finding the supremum amount of reward λ such that we would be indifferent on whether to play the standard arm or the calibrated arm. For a given arm, the model assumes the x_is is drawn from a parametric distribution indexed by θ with the density function $f(.|\Theta)$. The prior density for θ is denoted by π. The base for the calibration process is the Bellman equation rewritten as:

$$U(\lambda,\pi) = max\left[\frac{\lambda}{1-\gamma}, r(\pi) + \gamma \int U(\lambda,\pi_x) f(x|\pi)\, dx\right]$$

Where π_x denotes the posterior $\pi(\theta|x)$, and $f(.|\pi) = \int f(.|\theta)\pi(\theta)d\theta$.

2.4 Bernoulli Reward Process:

Most of the previous studies depicted that the human decision process by assuming Bernoulli Reward process. By this, we are assuming that the bandits generate sequences of independent and identically-distributed random values taking either R, with $R \in [0,100]$ fixed, or 0 with probability θ, and $1-\theta$, respectively. Without loss of generality we have assumed the value of R equals 1.

For a Bernoulli reward process, we will have a prior over θ as the Beta distribution with a density in the interval [0,1]

$$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\,\theta^{\alpha-1}\,(1-\theta)^{\beta-1}$$

Where $\alpha > 0\ and\ \beta > 0$, and the calibration equation becomes

$$U(\lambda,\alpha,\beta) = max\left[\frac{\lambda}{1-\gamma}, \frac{\alpha[1+\gamma U(\lambda,\alpha+1,\beta)] + \beta\gamma U(\lambda,\alpha,\beta+1)}{\alpha+\beta}\right]$$

3 Project Specification

The main objective of the project is to understand the model which depicts the human decision-making more appropriately. Author tried to generate the data of Human Sequential decisions through a game and compared with the data generated by different models like Gittins Index, Gittins Index with Memory restrictions, Bernoulli Reward Process without memory and look ahead restrictions and Bernoulli Reward processes with memory and look ahead restrictions. Author concludes that the Bernoulli Reward Process with memory and look ahead restrictions predict the human decisions better. By developing a model that describes the human behavior appropriately, we can train the agent to maximize the total expected reward.

To capture the human decisions, Author has designed a game with 4 bandits. At any given time, player can select one arm and rewarded with 0 or 1 with predefined θ and (1 - θ) probability respectively. Player is given finite number of chances to select the arms and with every selection of the arm, the chances are reduced. The game is stopped when there are no chances left to play.

Gittins Index is said to the optimal solution to MAB problems. But it assumes that agent learning the environment has infinite memory and infinite look ahead unlike the real scenario. Hence, author develops models that assumes agent has limited memory and look ahead and train the agent to pick the best arm to maximize the total reward.

4 Detailed Design

4.1 Game to capture Human Behavior

To capture Human Behavior, we have designed a game which creates a 4-arm multi-armed bandit environment.

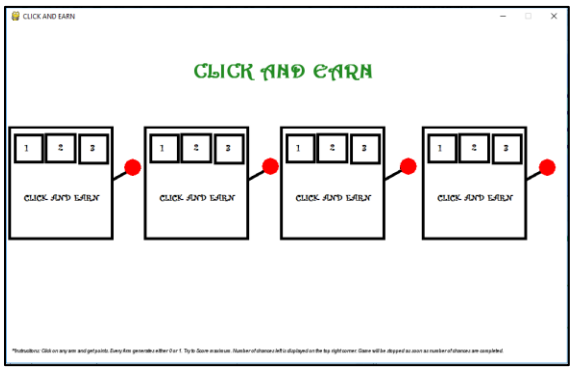

Figure 1: Game Start Screen

Each arm when selected generates a reward (either 0 or 1) with predefined probability. Player can select arms for limited number of times.

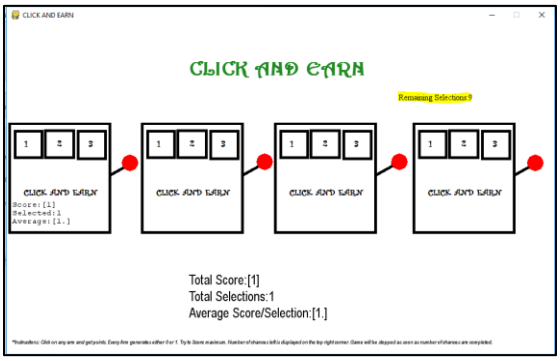

Figure 2: When Arm 1 is selected

For better user understandability, we have shown the total score, total number of selections done so far and the average reward per selection at the bottom of the game screen.

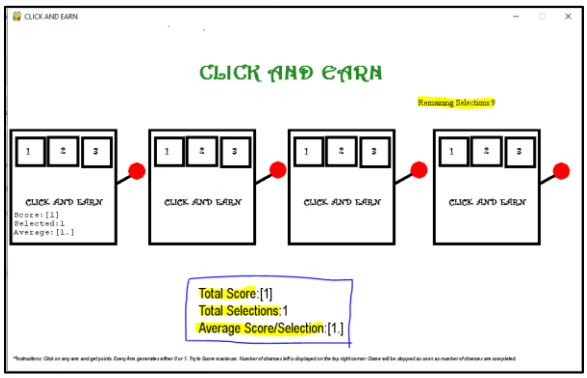

Figure 3: Score Details

Also, details like the number of times the bandit is selected, score obtained by selecting the bandit and the average score per selection for every bandit is selected.

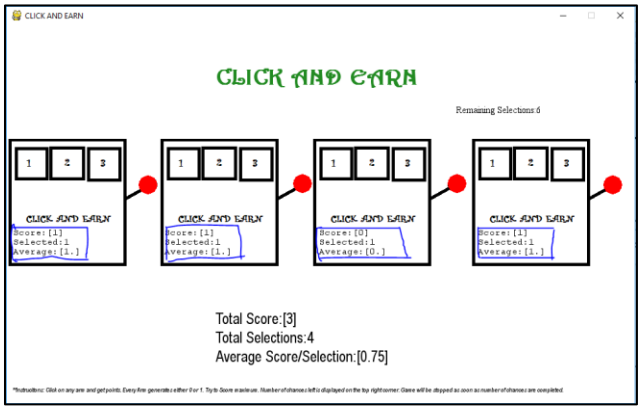

Figure 4: Game Screen with details of each bandit

Game starts as the first selection of the arm is made and the remaining number of chances is displayed at the top of the screen. As the chances given are completed, game ends and the scores are displayed.

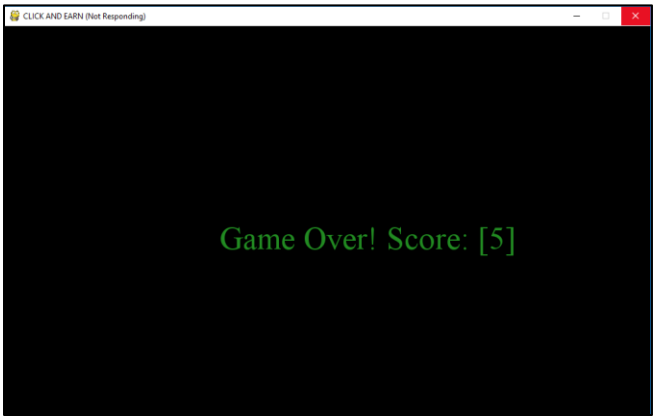

Figure 5: Game Stop Screen

The total number of chances and the score obtained is stored in the database for future analysis.

4.2 Modeling Human Belief Updates

As a part of the project, we try to develop below belief update models and find the best model that depicts human behavior with minimal difference.

- Gittins Index with Infinite Memory
- Gittins Index with Limited Memory
- Bernoulli Reward Process with Infinite Memory and Infinite Look Ahead
- Bernoulli Reward Process with Limited Memory and Limited Look Ahead

4.2.1 Gittins Index with Infinite Memory

This model chooses the arm with highest Gittins Index. Gittins Index is calculated by considering the total discounted reward of the arm. The discounted reward is obtained by considering the total rewards given by the arm till the current selection. The formula for calculating the Gittins Index is given in Section 2.3.

4.2.2 Gittins Index with Limited Memory

The Problem with previous accounts of Gittins Index as a model for human behavior is that they assume infinite memory and look ahead. In the paper "Bayesian Modeling of Human Sequential Decision-Making on the Multi-Armed Bandit problem", Acuna and Paul showed that models that restrict the memory usage can be more appropriate in depicting the human decision process and help in achieving greater rewards.

So, the equations for computing Gittins Index with restricted memory are as below:

$$G_i = \mathbb{E}_{\pi^i}\left[\sum_{t=\tau-m}^{\tau-1} \gamma^t\, r(\pi_i^t)\right] \Big/ \mathbb{E}_{\pi^i}\left[\sum_{t=\tau-m}^{\tau-1} \gamma^t\right] \ (Restricting\ memory\ )$$

And the calibration equation will be

$$U(\lambda, \pi, h) = max\left[\frac{\lambda}{1-\gamma}, r(\pi) + \gamma \int U(\lambda, \pi_x, h-1) f(x|\pi) dx\right] \ where\ h > 0 \ \ and$$

$$U(\lambda, \pi_x, 0) = S(\lambda, \pi_x)\ and\ S(\lambda, \pi) = max\left[\frac{\lambda}{1-\gamma}, \sum_{t=0}^{\infty} \gamma^t\, r(\pi)\right] (Restricting\ look\ ahead).$$

4.2.3 Bernoulli Reward Process with Infinite Memory and Infinite Look Ahead

This model chooses the arm with highest Gittins Index. Gittins Index is calculated using Calibration method. The rewards obtained by the arms are considered to the Bernoulli distributed. The Calibration equation used to compute the Gittins Index is given in Section 2.4

4.2.4 Bernoulli Reward Process with Limited Memory and Limited Look Ahead

After observing the sequence $x_1, x_2, \ldots\ldots x_n$ of rewards from the Bernoulli arm, the density function of the reward distribution changes to

$$f(x_i|\theta) = \theta^{x_i}(1-\theta)^{1-x_i}$$

Memory can be restricted by considering only the subsequence $x_{n-m}, x_{n-m+1}, \ldots\ldots\ldots x_n$ instead of entire sequence of rewards. Look ahead can be restricted by modifying the calibration equation as below

$$S(\lambda, \alpha, \beta) = max\,[\lambda, \alpha(\alpha+\beta)^{-1}]/(1-\gamma)$$

Where α is the number of occurrences of 1 and β is the number of occurrences of 0 in the subsequence of rewards $x_{n-m}, x_{n-m+1}, \ldots\ldots\ldots x_n$.

5 Experimental Results

By the experiments conducted, we observe that the *Bernoulli Reward Process with Limited Memory and look ahead* best depicts the human behavior. The scores are generated using the game for different number of episodes and similarly the score from proposed 4 models for different number of episodes is also generated and plotted to compare with the human behavior.

The memory is restricted to past 10 sequence of rewards instead of the entire past data. And the look ahead is also limited to 10 future rewards instead of entire sequence of rewards till stopping time.

Note: Author couldn't generate data for number of episodes greater than 20 for the third model by the time of writing of this report as the execution takes more time and memory.

Below graph is the comparison of data from the game and the 3 belief models (Section 4.2.1, 4.2.2, 4.2.4).

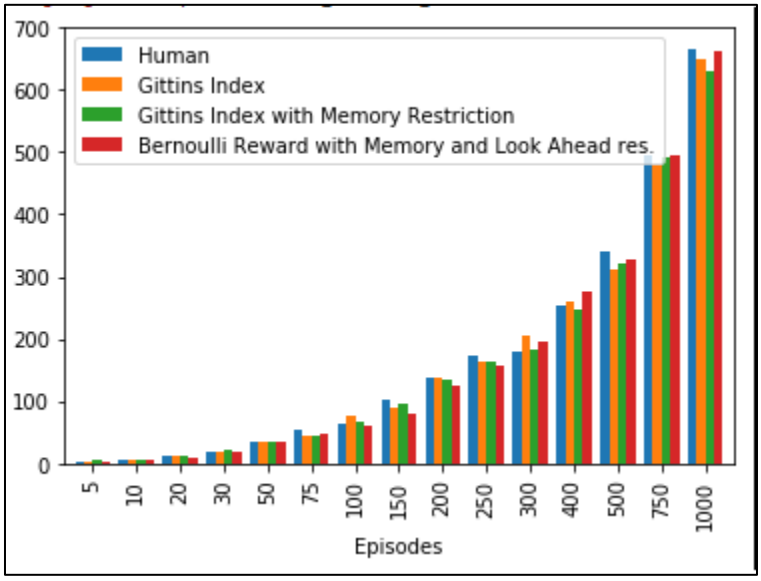

Figure 6: Comparison of Human Belief Models

6 References

[1] Acuna, DE & Schrater, P. (2008). Bayesian Modeling of Human Sequential Decision-Making on the Multi-Armed Bandit Problem. In V. Sloutsky, B. Love, & K. McRae (Eds.), Proceedings of the 30th Annual Conference of the Cognitive Science Society. Washington, DC: Cognitive Science.

[2] Richard S. Sutton and Andrew G. Barto, "Multi-arm Bandits", in "Reinforcement Learning: An Introduction" Second Edition, The MIT Press pg. 25-44

[3] Paul B. reverdy, Vaibhav Srivastava, Naomi Enrich Leonard, "Modeling Human Decision Making in Generalized Gaussian Multiarmed Bandits". In Conference: European Control Conference, At Strasbourg, France

[4] Peter Auer, Nicolo Cesa-Bianchi, Paul Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem". Proc. Of 15th International Conference on Machine Learning, 2002

[5] Anderson, Christopher Madden, "Behavioral models of strategies in multi-armed bandit problems"

[6] Banks, J., Oslon, M., & Porter, D. "An experimental analysis of the bandit problem" Economic Theory

[7] Wikipedia Contributors "Gittins Index", Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Gittins_index

[8] Wikipedia Contributors "Bernoulli Distribution", Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Bernoulli_distribution

[9] Sentdex, tutorials on pygame, https://pythonprogramming.net/displaying-images-pygame/

[10] Pandas Help Community, tutorials on Pandas DataFrames, https://pandas.pydata.org/pandas-docs/version/0.21/generated/pandas.DataFrame.html

7 Work Under Discussion

In the future author tries to extend this model to study of the behavior of a satisfier. Satisfiers are those with satisficing qualities unlike maximiser. Maximiser always tries to maximize the result. They always pick the best among the available options. Whereas satisfier will be happy above certain threshold level. Author will try to develop a model which maximizes the total reward when the agent chooses the arm that is closest to the satisficing arm by assuming that the agent has limited memory and look ahead.

8 Appendix A

8.1 Algorithm – Calculation of Gittins Index:

```
def playGame():
        Play all the arms once
        Calculate Gittins Index for all the arms
        Select the arm to with highest Gittins Index
        Repeat Step 2 and 3 till stopping time
End
```

8.2 Code

8.2.1 Game.py (Generates game environment)

```
import pygame
import numpy as np
import time
import pandas as pd
pygame.init() #initialize the python game
#Initialization of Global variables
displayWidth = 1000
displayHeight = 600 #Game Screen Dimensions


black = (0,0,0)
white = (255,255,255)
red = (255,0,0)
green = (34,139,34)
blue = (42,11,245)


click = 0 #to know the position of user click
totalReward = 0 #total score of the user
totalSelects = 10 #number of chances user is given
defaultReward = 0
individualReward = dict.fromkeys(['1','2','3','4'],defaultReward) #Hash to store individual rewards from arms
```

```
numOfSelects = dict.fromkeys(['1','2','3','4'],defaultReward)
probRewardBandit1 = [0.36,0.64]
probRewardBandit2 = [0.58,0.42]
probRewardBandit3 = [0.87,0.13]
probRewardBandit4 = [0.26,0.74]
possibleRewards = [0,1]

gameDisplay = pygame.display.set_mode((displayWidth,displayHeight)) #setting up game window
pygame.display.set_caption("CLICK AND EARN")
clock = pygame.time.Clock()
slotMach1 = pygame.image.load('slotmach1.png')
slotMach2 = pygame.image.load('slotmach1.png')
slotMach3 = pygame.image.load('slotmach1.png')
slotMach4 = pygame.image.load('slotmach1.png')

#Add the slot machine to display at particular location(x,y)
def slotMach(img,x,y):
    gameDisplay.blit(img,(x,y))

#funciton to generate points
def generatePoints(banditPicked):
    global totalReward,numOfSelects,individualReward,totalEpisodes
    if banditPicked == 1:
        individualReward['1'] = individualReward['1'] +
np.random.choice(possibleRewards,1,p=probRewardBandit1)
        numOfSelects['1'] = numOfSelects['1'] + 1
    elif  banditPicked == 2:
        individualReward['2'] = individualReward['2'] +
np.random.choice(possibleRewards,1,p=probRewardBandit2)
        numOfSelects['2'] = numOfSelects['2'] + 1
    elif banditPicked == 3:
        individualReward['3'] = individualReward['3'] +
np.random.choice(possibleRewards,1,p=probRewardBandit3)
```

```
        numOfSelects['3'] = numOfSelects['3'] + 1
    else:
        individualReward['4'] = individualReward['4'] +
np.random.choice(possibleRewards,1,p=probRewardBandit4)
        numOfSelects['4'] = numOfSelects['4'] + 1
    totalReward = sum(individualReward.values())
    totalEpisodes = sum(numOfSelects.values())

#print(totalReward,totalEpisodes,individualReward['1'],numOfSelects['1'],individualReward['2'],numOfSel
ects['2'],individualReward['3'],numOfSelects['3'],individualReward['4'],numOfSelects['4'])
def displayScoreBandit1():
    font = pygame.font.SysFont("monospace",15)
    #Bandit 1
    pygame.draw.rect(gameDisplay,white,(10,315,170,50),0)
    ScoreBandit1 = font.render("Score:"+str(individualReward['1']),1,black)
    gameDisplay.blit(ScoreBandit1,(10,320))
    SelectsBandit1 = font.render("Selected:"+str(numOfSelects['1']),1,black)
    gameDisplay.blit(SelectsBandit1,(10,335))
    avgBandit1 =
font.render("Average:"+str(np.round_(individualReward['1']/numOfSelects['1'],2)),1,black)
    gameDisplay.blit(avgBandit1,(10,350))
    pygame.display.update()

def displayScoreBandit2():
    #Bandit 2
    font = pygame.font.SysFont("monospace",15)
    pygame.draw.rect(gameDisplay,white,(251,315,170,50),0)
    ScoreBandit2 = font.render("Score:"+str(individualReward['2']),1,black)
    gameDisplay.blit(ScoreBandit2,(251,320))
    SelectsBandit2 = font.render("Selected:"+str(numOfSelects['2']),1,black)
    gameDisplay.blit(SelectsBandit2,(251,335))
    avgBandit2 =
font.render("Average:"+str(np.round_(individualReward['2']/numOfSelects['2'],2)),1,black)
```

```
    gameDisplay.blit(avgBandit2,(251,350))
    pygame.display.update()

def displayScoreBandit3():
    #Bandit 3
    font = pygame.font.SysFont("monospace",15)
    pygame.draw.rect(gameDisplay,white,(497,315,170,50),0)
    ScoreBandit3 = font.render("Score:"+str(individualReward['3']),1,black)
    gameDisplay.blit(ScoreBandit3,(497,320))
    SelectsBandit3 = font.render("Selected:"+str(numOfSelects['3']),1,black)
    gameDisplay.blit(SelectsBandit3,(497,335))
    avgBandit3 =
font.render("Average:"+str(np.round_(individualReward['3']/numOfSelects['3'],2)),1,black)
    gameDisplay.blit(avgBandit3,(497,350))
    pygame.display.update()

def displayScoreBandit4():
    #Bandit 4
    font = pygame.font.SysFont("monospace",15)
    pygame.draw.rect(gameDisplay,white,(750,315,170,50),0)
    ScoreBandit4 = font.render("Score:"+str(individualReward['4']),1,black)
    gameDisplay.blit(ScoreBandit4,(750,320))
    SelectsBandit4 = font.render("Selected:"+str(numOfSelects['4']),1,black)
    gameDisplay.blit(SelectsBandit4,(750,335))
    avgBandit4 =
font.render("Average:"+str(np.round_(individualReward['4']/numOfSelects['4'],2)),1,black)
    gameDisplay.blit(avgBandit4,(750,350))
    pygame.display.update()

def displayTotalScore():
    font = pygame.font.SysFont("arial",25)
    pygame.draw.rect(gameDisplay,white,(displayWidth/3,450,480,100),0)
```

```
    TotalScore = font.render("Total Score:"+str(sum(individualReward.values())),1,black)
    gameDisplay.blit(TotalScore,(displayWidth/3,450))
    TotalSelects = font.render("Total Selections:"+str(sum(numOfSelects.values())),1,black)
    gameDisplay.blit(TotalSelects,(displayWidth/3,480))
    TotalAverage = font.render("Average
Score/Selection:"+str(np.round_(sum(individualReward.values())/sum(numOfSelects.values()),2)),1,black)
    gameDisplay.blit(TotalAverage,(displayWidth/3,510))
    pygame.display.update()


def displaySelects():
    font = pygame.font.SysFont("Serif",15)
    pygame.draw.rect(gameDisplay,white,(715,125,500,50),0)
    TotalScore = font.render("Remaining Selections:"+str(totalSelects -
sum(numOfSelects.values())),1,black)
    gameDisplay.blit(TotalScore,(715,125))
    pygame.display.update()


def savescore():
    dataFrame =
pd.DataFrame(data={'Episodes':[totalSelects],'Score':[int(sum(individualReward.values()))]})
    print(dataFrame)
    dataFrame.to_csv("rewards.csv",mode="a",header=False,index=False)
def startGame():
    stopGame = False #variable used to stop the game
    x1 = 1
    y1 = (displayHeight * 0.3)
    x2 = 243
    y2 = (displayHeight * 0.3)
    x3 = (243 * 2)
    y3 = (displayHeight * 0.3)
    x4 = ((243 * 3) + 10)
    y4 = (displayHeight * 0.3) #coordinates for slot machines position
```

```
gameDisplay.fill(white)

#title display

font = pygame.font.SysFont("harrington",35,bold=True)

pygame.draw.rect(gameDisplay,white,(displayWidth/3,60,425,95),0)

gameTitle = font.render("CLICK AND EARN",1,green)

gameDisplay.blit(gameTitle,(displayWidth/3,60))

#instructions display

font = pygame.font.SysFont("arial",10,italic=True)

pygame.draw.rect(gameDisplay,white,(9,572,950,15),0)

gameTitle = font.render("**Instrucitons: Click on any arm and get points. Every Arm generates either 0 or 1. Try to Score maximum. Number of chances left is displayed on the top right corner. Game will be stopped as soon as number of chances are completed.",1,black)

gameDisplay.blit(gameTitle,(9,572))

#slot machines display

slotMach(slotMach1,x1,y1)

slotMach(slotMach2,x2,y2)

slotMach(slotMach3,x3,y3)

slotMach(slotMach4,x4,y4)

#Buttons

buttonBandit1 = pygame.draw.circle(gameDisplay,red,(225,253),15)

buttonBandit2 = pygame.draw.circle(gameDisplay,red,(471,251),15)

buttonBandit3 = pygame.draw.circle(gameDisplay,red,(717,251),15)

buttonBandit4 = pygame.draw.circle(gameDisplay,red,(965,253),15)

pygame.display.update()

while not stopGame:

    for event in pygame.event.get():

        if event.type == pygame.QUIT:

            stopGame = True

        if (totalSelects - sum(numOfSelects.values())) == 0:

            gameDisplay.fill(black)

            font = pygame.font.SysFont("Serif",50)

            pygame.draw.rect(gameDisplay,black,((displayWidth/3),(displayHeight/2),250,50),0)
```

```
                TotalScore = font.render("Game Over! Score: "+str(sum(individualReward.values())),1,green)
                gameDisplay.blit(TotalScore,((displayWidth/3),(displayHeight/2)))
                pygame.display.update()
                time.sleep(1)
                stopGame = True
            if event.type == pygame.MOUSEBUTTONDOWN:
                print(pygame.mouse.get_pos())
                if buttonBandit1.collidepoint(pygame.mouse.get_pos()) == 1:
                    generatePoints(1)
                    displayScoreBandit1()
                    displayTotalScore()
                    displaySelects()
                elif buttonBandit2.collidepoint(pygame.mouse.get_pos()) == 1:
                    generatePoints(2)
                    displayScoreBandit2()
                    displayTotalScore()
                    displaySelects()
                elif buttonBandit3.collidepoint(pygame.mouse.get_pos()) == 1:
                    generatePoints(3)
                    displayScoreBandit3()
                    displayTotalScore()
                    displaySelects()
                elif buttonBandit4.collidepoint(pygame.mouse.get_pos()) == 1:
                    generatePoints(4)
                    displayScoreBandit4()
                    displayTotalScore()
                    displaySelects()
#main loop
startGame()
savescore()
pygame.quit()
```

8.2.2 Gittins_Index.py (Model 1)

```
import numpy as np

import pandas as pd


#initialization

totalBandit = 4

totalSelects = 1000

defaultReward = 0

defaultIndex = 0

discountFactor = 0.36

discountedReward = 0

discountedFactor = 0


numOfSelects = dict.fromkeys([0,1,2,3],defaultReward)

individualTotalReward = dict.fromkeys([0,1,2,3],defaultReward)

individualReward = [ [defaultReward] * totalSelects for i in range(totalBandit) ]

gittinIndex = dict.fromkeys([0,1,2,3],defaultIndex)

memoryLimitation = 10


probRewardBandit1 = [0.36,0.64]

probRewardBandit2 = [0.58,0.42]

probRewardBandit3 = [0.87,0.13]

probRewardBandit4 = [0.26,0.74]

possibleRewards = [0,1]


def saveScore(file_name):

   dataFrame =
pd.DataFrame(data={'Episodes':[totalSelects],'Score':[int(sum(individualTotalReward.values()))]})

   print(dataFrame)

   dataFrame.to_csv(file_name,mode="a",header=False,index=False)


def generateReward(banditSelected):
```

```
    if banditSelected == 0:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit1)
    elif banditSelected == 1:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
    elif banditSelected == 2:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
    elif banditSelected == 3:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit3)


    return int(reward)
def selectBandit():
    global discountedReward,discountedFactor
    for i in range(totalBandit):
        discountedReward = 0
        discountedFactor = 0
        for j in range(numOfSelects[i]):
            discountedReward = discountedReward + ((discountFactor ** j) * individualReward[i][j])
            discountedFactor = discountedFactor + (discountFactor ** j)
        gittinIndex[i] = (discountedReward/discountedFactor)
    return max(gittinIndex,key=gittinIndex.get)
def playGame():
    for i in range(totalBandit):
        banditSelected = i
        numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
        pointsRewarded = generateReward(banditSelected)
        individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
        individualReward[int(banditSelected)][numOfSelects[banditSelected]-1] = pointsRewarded
    stopGame = False
    while not stopGame:
        if totalSelects - sum(numOfSelects.values()) == 0:
            stopGame = True
```

```
        else:
            banditSelected = selectBandit()
            numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
            pointsRewarded = generateReward(banditSelected)
            individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
            individualReward[banditSelected][numOfSelects[banditSelected]-1] = pointsRewarded
    #Main loop
    playGame()
    saveScore('rewards_gittinIndex.csv')
```

8.2.3 gittinsIndex_withLimitedMemory.py (Model 2)

```
    import numpy as np
    import pandas as pd
    #initialization
    totalBandit = 4
    totalSelects = 1000
    defaultReward = 0
    defaultIndex = 0
    discountFactor = 0.36
    discountedReward = 0
    discountedFactor = 0

    numOfSelects = dict.fromkeys([0,1,2,3],defaultReward)
    individualTotalReward = dict.fromkeys([0,1,2,3],defaultReward)
    individualReward = [ [defaultReward] * totalSelects for i in range(totalBandit) ]
    gittinIndex = dict.fromkeys([0,1,2,3],defaultIndex)
    memoryLimitation = 10

    probRewardBandit1 = [0.36,0.64]
    probRewardBandit2 = [0.58,0.42]
    probRewardBandit3 = [0.87,0.13]
    probRewardBandit4 = [0.26,0.74]
    possibleRewards = [0,1]
```

```
def saveScore(file_name):
    dataFrame = pd.DataFrame(data={'Episodes':[totalSelects],'Score':[int(sum(individualTotalReward.values()))]})
    print(dataFrame)
    dataFrame.to_csv(file_name,mode="a",header=False,index=False)


def generateReward(banditSelected):
    if banditSelected == 0:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit1)
    elif banditSelected == 1:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
    elif banditSelected == 2:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
    elif banditSelected == 3:
        reward = np.random.choice(possibleRewards,1,p=probRewardBandit3)
    return int(reward)
def selectbanditWithLimitedMemory():
    global discountedReward,discountedFactor
    for i in range(totalBandit):
        discountedReward = 0
        discountedFactor = 0
        if numOfSelects[i] >= memoryLimitation:
            number = memoryLimitation
        else:
            number = numOfSelects[i]
        for j in range(number):
            discountedReward = discountedReward + ((discountFactor ** j) * individualReward[i][numOfSelects[i]-j])
            discountedFactor = discountedFactor + (discountFactor ** j)
        gittinIndex[i] = (discountedReward/discountedFactor)
    return max(gittinIndex,key=gittinIndex.get)
```

```
def playGameWithLimitedMemory():
    #re initializing because previous values must be cleared
    for i in range(totalBandit):
        banditSelected = i
        numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
        pointsRewarded = generateReward(banditSelected)
        individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
        individualReward[int(banditSelected)][numOfSelects[banditSelected]-1] = pointsRewarded
    stopGame = False
    while not stopGame:
        if totalSelects - sum(numOfSelects.values()) == 0:
            stopGame = True
        else:
            banditSelected = selectbanditWithLimitedMemory()
            numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
            pointsRewarded = generateReward(banditSelected)
            individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
            individualReward[banditSelected][numOfSelects[banditSelected]-1] = pointsRewarded


#Main loop
playGameWithLimitedMemory()
saveScore('rewards_gittinsIndex_withLimitedmemory.csv')
```

8.2.4 Bernoulli_rewards-without_restrictions.py (Model 3)

```
import pandas as pd
import numpy as np
#initialization
totalBandit = 2
totalSelects = 1
defaultReward = 0
defaultIndex = 0
discountFactor =0.36
discountedReward = 0
```

```
discountedFactor = 0

standardReward = 1


numOfSelects = dict.fromkeys([0,1,2,3],defaultReward)

individualTotalReward = dict.fromkeys([0,1,2,3],defaultReward)

individualReward = [ [defaultReward] * totalSelects for i in range(totalBandit) ]

gittinIndex = dict.fromkeys([0,1,2,3],defaultIndex)

memoryLimitation = 10


probRewardBandit1 = [0.36,0.64]

probRewardBandit2 = [0.58,0.42]

probRewardBandit3 = [0.87,0.13]

probRewardBandit4 = [0.26,0.74]

possibleRewards = [0,1]


def saveScore(file_name):

    dataFrame                                                                    =
pd.DataFrame(data={'Episodes':[totalSelects],'Score':[int(sum(individualTotalReward.values()))]})

    print(dataFrame)

    dataFrame.to_csv(file_name,mode="a",header=False,index=False)


def generateReward(banditSelected):

  if banditSelected == 0:

    reward = np.random.choice(possibleRewards,1,p=probRewardBandit1)

  elif banditSelected == 1:

    reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)

  elif banditSelected == 2:

    reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)

  elif banditSelected == 3:

    reward = np.random.choice(possibleRewards,1,p=probRewardBandit3)


  return int(reward)
```

```
def calcGittinsIndex(bandit):
  discountedReward = 0
  discountedFactor = 0
  for j in range(totalSelects):
    discountedReward = discountedReward + ((discountFactor ** j) * individualReward[bandit][j])
    discountedFactor = discountedFactor + (discountFactor ** j)
  ratio = (discountedReward/discountedFactor)
  return ratio


def calculateGittinsIndex(bandit,standardReward,alpha,beta):
    #print("Alpha,Beta:",alpha,beta)
    if ((alpha + beta) == totalSelects):
      return calcGittinsIndex(bandit)
    else:
        first_part = standardReward / ( 1 - discountFactor )
        second_part = ((alpha * ( 1 + (discountFactor * calculateGittinsIndex(bandit,standardReward,alpha+1,beta)))) + beta * discountFactor * calculateGittinsIndex(bandit,standardReward,alpha,beta+1)) / (alpha + beta)
    if first_part > second_part:
        return first_part
    elif second_part > first_part:
        return second_part
    else:
        return standardReward


def selectBandit():
    for i in range(totalBandit):
        gittinIndex[i] = calculateGittinsIndex(i,standardReward,individualTotalReward[i],numOfSelects[i]-individualTotalReward[i])
    return max(gittinIndex,key=gittinIndex.get)


def playGame():
```

```
    for i in range(totalBandit):
        banditSelected = i
        #print("Bandit Selected",i)
        numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
        pointsRewarded = generateReward(banditSelected)
        #print("Points Awarded",pointsRewarded)
        individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
        individualReward[int(banditSelected)][numOfSelects[banditSelected]-1] = pointsRewarded


    stopGame = False
    while not stopGame:
        if totalSelects - sum(numOfSelects.values()) == 0:
            stopGame = True
        else:
            banditSelected = selectBandit()
            #print("Bandit Selected",banditSelected)
            numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
            pointsRewarded = generateReward(banditSelected)
            #print("Points Awarded",pointsRewarded)
            individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
            individualReward[banditSelected][numOfSelects[banditSelected]-1] = pointsRewarded
playGame()
saveScore('rewards_bernoulli.csv')
```

8.2.5 Bernoulli_rewards.py

```
import pandas as pd
import numpy as np
#initialization
totalBandit = 4
totalSelects = 1000
defaultReward = 0
defaultIndex = 0
discountFactor =0.36
```

```
discountedReward = 0
discountedFactor = 0
standardReward = 1

numOfSelects = dict.fromkeys([0,1,2,3],defaultReward)
individualTotalReward = dict.fromkeys([0,1,2,3],defaultReward)
individualReward = [ [defaultReward] * totalSelects for i in range(totalBandit) ]
gittinIndex = dict.fromkeys([0,1,2,3],defaultIndex)
memoryLimitation = 10
lookAhead = 10

probRewardBandit1 = [0.36,0.64]
probRewardBandit2 = [0.58,0.42]
probRewardBandit3 = [0.87,0.13]
probRewardBandit4 = [0.26,0.74]
possibleRewards = [0,1]

def saveScore(file_name):
    dataFrame = pd.DataFrame(data={'Episodes':[totalSelects],'Score':[int(sum(individualTotalReward.values()))]})
    print(dataFrame)
    dataFrame.to_csv(file_name,mode="a",header=False,index=False)

def generateReward(banditSelected):
  if banditSelected == 0:
    reward = np.random.choice(possibleRewards,1,p=probRewardBandit1)
  elif banditSelected == 1:
    reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
  elif banditSelected == 2:
    reward = np.random.choice(possibleRewards,1,p=probRewardBandit2)
  elif banditSelected == 3:
    reward = np.random.choice(possibleRewards,1,p=probRewardBandit3)
```

```
    return int(reward)
def calcGittinsIndex(bandit,alpha,beta):
    first_part = standardReward/(1-discountFactor)
    second_part = (alpha/(alpha + beta))/(1 - discountFactor)
    if first_part > second_part:
        return first_part
    elif second_part > first_part:
        return second_part
    else:
        return standardReward


def calculateGittinsIndex(bandit,standardReward,alpha,beta,h):
    #print(alpha,beta)
    if (h == 0):
      return calcGittinsIndex(bandit,alpha,beta)
    else:
        first_part = standardReward / ( 1 - discountFactor )
        second_part = ((alpha * ( 1 + (discountFactor * calculateGittinsIndex(bandit,standardReward,alpha+1,beta,h-1)))) + beta * discountFactor * calculateGittinsIndex(bandit,standardReward,alpha,beta+1,h-1)) / (alpha + beta)
    if first_part > second_part:
        return first_part
    elif second_part > first_part:
        return second_part
    else:
        return standardReward


def selectBandit():
    for i in range(totalBandit):
        gittinIndex[i] = calculateGittinsIndex(i,standardReward,individualTotalReward[i],numOfSelects[i]-individualTotalReward[i],lookAhead)
    return max(gittinIndex,key=gittinIndex.get)
def playGame():
```

```
    for i in range(totalBandit):
        banditSelected = i
        numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
        pointsRewarded = generateReward(banditSelected)
        individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
        individualReward[int(banditSelected)][numOfSelects[banditSelected]-1] = pointsRewarded
    stopGame = False
    while not stopGame:
        if totalSelects - sum(numOfSelects.values()) == 0:
            stopGame = True
        else:
            banditSelected = selectBandit()
            numOfSelects[banditSelected] = numOfSelects[banditSelected] + 1
            pointsRewarded = generateReward(banditSelected)
            individualTotalReward[banditSelected]= individualTotalReward[banditSelected] + pointsRewarded
            individualReward[banditSelected][numOfSelects[banditSelected]-1] = pointsRewarded
playGame()
saveScore('rewards_bernoulli_MemLim_LALim.csv')
```

9 Appendix B

The data generated using the game and models are as below:

Episodes	Human Score	Model 1 Score	Model 2 Score	Model 4 Score
5	2	4	2	2
10	7	6	4	5
20	11	12	12	9
30	17	21	19	18
50	33	33	35	33
75	53	45	43	48
100	64	67	76	61
150	103	96	89	78
200	139	135	139	123
250	172	163	163	156
300	179	181	204	194
400	255	247	259	276
500	341	320	311	326
750	496	492	479	495
1000	667	629	650	663

Table 1: Comparison of Episodes-Scores by different models

10 Appendix C

Annotated Bibliography

[1] Paul B. reverdy, Vaibhav Srivastava, Naomi Enrich Leonard, "Modeling Human Decision Making in Generalized Gaussian Multiarmed Bandits". In Conference: European Control Conference, At Strasbourg, France

This paper attempts to build models and confidence limits for the Human Decision-Making Features assuming they follow Gaussian distribution. They model 5 salient human decision-making features like Familiarity with the environment, Ambiguity Bonus, Stochasticity, Finite-horizon effects, Environmental Structural effects. This paper shows that Human Decisions are efficiently captured by the stochastic UCL algorithm with appropriate parameters.

[2] Daniel Acuña, Paul Schrater, "Bayesian modeling of Human Sequential Decision-Making on the Multi-Armed Bandit Problem". In V. Sloutsky, B. Love, & K. McRae (Eds.), Proceedings of the 30th Annual Conference of the Cognitive Science Society. Washington, DC

This paper attempts to investigate human exploration/exploitation behavior in sequential-decision making tasks. By incorporating more realistic assumptions about subject's knowledge and limitations into models of belief updating, this paper shows that Bayesian models of Human Behavior for the Multi-Armed Bandit Problems perform better than previous studies.

[3] Paul B. Reverdy, Vaibhav Srivastava, Naomi Enrich Leonard, "Satisficing in Multi-Armed Bandit problems" Published 2017 in IEEE Transactions on Automatic Control.

This paper helped me in my research work related to satisficing. It proposes two sets of satisficing objectives for the multi-armed bandit problem, where the objective is to achieve reward-based decision-making performance above a given threshold. It shows that the new objectives defined are equivalent to standard bandit problems and use them to find the optimal bounds. They didn't speak about the memory limitation which wonders me.

[4] Mohammand Abu, Alsheikh, Dinh Thai Hoang, Dusit Niyato, Hwee-Pink Tan, Shaowei Lin, "Markov Decision Processes with Applications in Wireless Sensor Networks: A Survey"

This is a survey paper about the applications of Markov Decisions Process in Wireless Sensor networks. This is the first research paper that I ever read. It introduced me to the vast applications of Markov Decision Process in the field of Sensor Networks. It explains different problems in Sensor Networks, the solutions already available and how MDP solves them better and efficiently.

[5] Xin Fei, Azzedine Boukerche, Richard Yu, "An Efficient Markov Decision Process Based Mobile Data Gathering Protocol for Wireless Sensor Networks"

This paper provides a solution to improve performance of data gathering while saving energy in wireless sensor networks. It compares the traditional Virtual Force method of data gathering with the solution using MDP. They show the results where MDP based Algorithm outperforms than all other traditional algorithms.

[6] Peter Auer, Nicolo Cesa-Bianchi, Paul Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem". Proc. Of 15th International Conference on Machine Learning, 2002

This paper helped me in understanding the regret analysis in multiarmed bandit problems. It says that the total expected regret grows at least logarithmically in the number of plays.

[7] Anderson, Christopher Madden, "Behavioral models of strategies in multi-armed bandit problems"

This paper describes different behavioral models that depict human behavior using multi-arm bandit framework. They say that ambiguity aversion is likely cause of suboptimal play in bandits.

[8] Banks, J., Oslon, M., & Porter, D. "An experimental analysis of the bandit problem" Economic Theory

This chapter explains the behavior of single arm bandit problems with finite set of options, discrete time intervals over an infinite horizon. They also talk about two arm bandit problems and how they differ with theoretical results quantitatively.

[9] J. -Y Audibert, R Munos and C. Szepesvari, "Exploration/Exploitaion tradeoff using variance estimates in multi-armed bandits"

This paper considers a variant of the basic algorithm for the stochastic, multi-armed bandit problem that considers the empirical variance of the different arms. They show that the algorithm that uses the variance estimates has a major advantage over its alternatives that do not use such estimates provided that the variances of the payoffs of the suboptimal arms are low. They also prove that the regret concentrates only at a polynomial rate.

[10] Pannagadatta Shivaswamy and Thorsten Joachims, "Multi-armed Bandit Problems with History"

This paper the history about the arms that is available prior to applying the algorithm to find which arm to select. They propose algorithms (Naïve algorithm) that show logarithmic amount of historic data allows them to achieve constant regret. They also determine upper confidence bounds with historic data for multi-armed bandit problems.

[11] Joann`es Vermorel and Mehryar Mohri, "Multi-Armed Bandit Algorithms and Empirical Evaluation"

This paper describes methods to evaluate alogorithms that are proposed to solve the Multi-Armed Bandit Problem. They also propose a new algorithm POKER (price of knowledge and Estimated Reward) to solve the same. They also conclude that ϵ-greedy is the best algorithm till date.

[12] Volodymyr Kuleshov, Doina Precup," Algorithms for the multi-armed bandit problem"

This paper explains different algorithms to solve multi-armed bandit problems and determine their effectiveness in solving the problem. They conclude that simple heuristics like Boltzmann exploration and ϵ-greedy outperform theoretically sound algorithms.

[13] Adam J. Mersereau, Paat Rusmevichientong, and John N. Tsitsiklis, "A Structured Multiarmed Bandit Problem and the Greedy Policy"

This paper focus on the effectiveness of greedy algorithms in solving the multi armed bandit problems. They prove that in the infinite horizon discounted reward setting, they show that the greedy and optimal policies eventually coincide, and both settle on the best arm.

[14] Esther Levin, Roberto Pieraccini, Wieland Eckert, "Using Markov Decision Process for Learning Dialogue Strategies"

This paper attempts to learn dialogue strategy for an air travel information system. They introduce a stochastic model for dialogue systems based in Markov Decision process.

[15] Tommi Jaakkola, Satinder P. Singh, Michael I. Jordan, "Reinforcement Learning Algorithm for Partially Observable Markov Decision Problems"

This paper explains Partially Observable Markov Decision Processes. They develop algorithm that applies to solve a certain class of Non-Markov decision problems.

[16] Richard S. Sutton, David McAllester, Satinder Singh, Yishay Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation"

This paper explores an alternative approach in which the policy is explicitly represented by its own function approximator, independent of the value function, and is updated according to the gradient of expected reward with respect to the policy parameters. This paper helped me to get better knowledge on the Policy Gradient methods.