ORF418 Final Project: Ocean's 418

Connie Zhu
Woramanot Yomjinda

Part I: Narrative

A casino would like to detect card counters in Blackjack over N rounds. Players can be of two types: non card counters (who win with probability .40) and card counters (who win with probability .52). While the true probabilities of winning in Blackjack for non card counters is closer to .50 if they play with optimal strategies, we slightly exaggerate the difference in our model, to see the difference in results for a shorter window of time.

In our model, the game starts at time 0 and ends at time N, where one person plays the game in each round. During the round, the casino observes the bet size and win/loss of the player. Before the player places a bet (between 0 and 1), the casino can decide to either ban the player or let them continue. If the casino bans the player, then a new player (independent from previous players) comes to play. Whenever a new player comes to the table (at the beginning or after we ban the previous player), we generate a new average bet size attribute $\bar{a}$ for a new player for the next round.

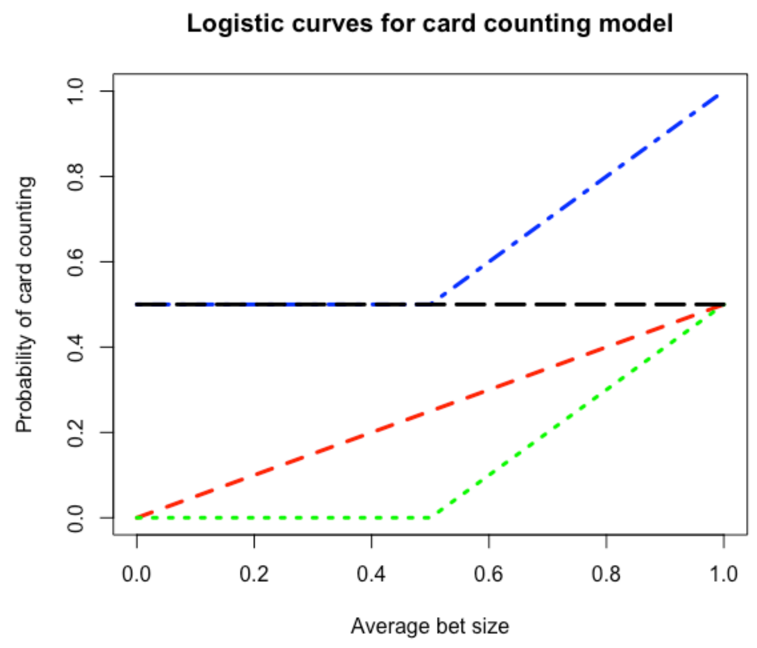
Part II: Mathematical Model

<u>Introduction:</u>

Our belief model is parametric, with four possible beliefs $\overrightarrow{\theta_\iota}$ (i from 1 to 4) that may be the truth. Let C be an indicator variable for if the player is a card counter. The logistic curves are defined as $P(C=1) = \min\left(<\vec{\theta}, \vec{a}>^{+}, 1\right) = \min((\theta^1 + \theta^2 \bar{\text{a}})^{+},\ 1)$ where $<\vec{\theta}, \vec{a}>$ is the dot product of vector $\vec{\theta} = \begin{matrix}\theta^1 \\ \\ \theta^2\end{matrix}$, a vector of true parameters for our logistic curve and $\overrightarrow{a} = \begin{matrix}1 \\ \\ \bar{\text{a}}\end{matrix}$, where a is the average bet size attribute, distributed $\sim N(\bar{a}, \sigma_a^2),\ \beta = \frac{1}{\sigma_a^2}$, where $\bar{a}$ is the true average bet size.

Our four logistic curves:

- $\overrightarrow{\theta_1} = <0, \frac{1}{2}>$. This belief (red line) proposes that probability of card counting is linearly increasing with the average bet size. Card counters, who know that they have positive expected value from the game, will choose to increase their wealth by betting larger amounts than non-card counters, especially if they are myopic and care more about near-term winnings compared to possibly alerting the casino of their type.
- $\overrightarrow{\theta_2} = <-\frac{1}{2}, 1>$. This belief (green line) corresponds to the case where higher average bet size correlates more strongly to the card counting type, but only above the threshold a = 1/2. That is, the casino is sure that for players with attribute a < ½, they are not card counters.
- $\overrightarrow{\theta_3} = <\frac{1}{2}, \frac{1}{2}>$. Opposite belief 2, this belief (blue line) corresponds to the case where the casino is sure that players with a > ½ are card counters.
- $\vec{\theta}_4 = <\frac{1}{2}, 0>$. This belief (black line) corresponds to the case where the card counter type is not correlated to the average bet size (i.e. card counters do not have a greater bet size than non card counters). In real life, card counters can disguise their type from the casino by not drawing attention to themselves through observable traits such as larger bet sizes.



Graphical representation of our logistic curves.

In simulation, when a new player comes to the table, we randomly generate his true $\bar{a}$. Then, we input $\bar{a}$ into the true parametric curve $\vec{\theta}$, which gives us P(C = 1). We then decide that this player is a card counter with probability P(C = 1) and non card counter with probability 1 – P(C = 1). This truth is hidden from the casino.

<u>State variables</u>:

We define the state variables at the iteration n to be

$S^n = (\vec{q}, \bar{a}, \beta\ ) = (q_1^n, q_2^n, q_3^n, q_4^n, \bar{a}^n\ , \beta^n)$

q_i^n: probability at round n that belief i is the truth

$\bar{a}^n$: estimate at round n of current player's true average bet size $\bar{a}$.

β^n: precision of the $\bar{a}^n$ estimate for $\bar{a}$

For our prior, we assume that each of the four beliefs are equally likely to be the truth. Since players are independent from each other, when a player has just arrived at the table, the casino uses their priors for $\bar{a}^0, \beta^0$, values based on records of past players.

$S^0 = \left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \bar{a}^0, \beta^0\right)$

<u>Decision variables</u>:

$x^n = x^\pi(S^n),\ x^n \in \{0,1\}$

At round n, we use a policy to generate our decision variable x^n to either ban the player (x^n = 1) or let them continue (x^n = 0). If we ban them, a new player (independent from previous players) comes in next round.

New Information:

At time n, the house observes W^{n+1} which contains the player's bet size and the indicator variable for whether the person wins at time n.

$$W^{n+1} = (\hat{A}^{n+1}, \hat{B}^{n+1})$$

$\hat{A}^{n+1}$ is the bet amount made by the current player, where $\hat{A}^{n+1} \in [0,1]$ due to the casino's imposed bet ceiling and floor of 0 and 1, respectively.
$\hat{B}^{n+1}$ is the indicator variable for whether the player wins or loses the round.

Transition function:

$S^{n+1} = S^M(S^n, x^n, W^{n+1})$
If the casino lets the player continue $(x^n = 0)$, *we use Bayesian updating for our estimate of* a^{n+1}.

$$\bar{a}^{n+1} = \frac{\beta^n \bar{a}^n + \beta^W \hat{A}^{n+1}}{\beta^n + \beta^W}$$

$$\beta^{n+1} = \beta^n + \beta^W$$

If the casino bans the player $(x^n = 1)$, *we set* $\bar{a}^{n+1} = \bar{a}^0$ *and* $\beta^{n+1} = \beta^0$ (*priors*).

We update $\vec{q}$ according to a weighted fraction of multiplying the probability that the player got the outcome $\hat{B}^{n+1}$ under belief i and attribute $\bar{a}^{n+1}$ by the probability that belief i is the truth.

$$q_i^{n+1} = \frac{q_i^n P(\hat{B}^{n+1} | \theta = \theta_i, \bar{a}^{n+1})}{\sum_{i=1}^4 q_i^n P(\hat{B}^{n+1} | \theta = \theta_i, \bar{a}^{n+1})}$$

where $P(\hat{B}^{n+1} | \theta = \theta_i, \bar{a}^{n+1}) = P(\hat{B}^{n+1}, C = 1 | \theta = \theta_i, \bar{a}^{n+1}) + P(\hat{B}^{n+1}, C = 0 | \theta = \theta_i, \bar{a}^{n+1})$

$$= P(\hat{B}^{n+1} | C = 1, \theta = \theta_i, \bar{a}^{n+1}) * P(C = 1 | \theta = \theta_i, \bar{a}^{n+1}) + P(\hat{B}^{n+1} | C = 0, \theta = \theta_i, \bar{a}^{n+1}) * (1 - P(C = 1 | \theta = \theta_i, \bar{a}^{n+1}))$$

$$= \{0.52 \; if \; \hat{B}^{n+1} = 1, 0.48 \; if \; \hat{B}^{n+1} = 0\} * P(C = 1 | \theta = \theta_i, \bar{a}^{n+1}) + \{0.4 \; if \; \hat{B}^{n+1} = 1, 0.6 \; if \; \hat{B}^{n+1} = 0\} * (1 - P(C = 1 | \theta = \theta_i, \bar{a}^{n+1}))$$

Objective Function:

The house wants to maximize expected revenue (i.e. minimize the player's expected winnings) over a given policy from time 1 to N. Since we are concerned with the cumulative profit, our problem falls under online learning.

The performance metric is given as
revenue for period n = $C(S^n, X^\pi(S^n), W^{n+1}) = \hat{A}^{n+1} * (2 * \hat{B}^{n+1} - 1)$

The Objective is

$$\max_\pi \mathbb{E} \sum_{n=1}^{N} C(S^n, X^\pi(S^n), W^{n+1}) = \max_\pi \mathbb{E}\left[\sum_{n=1}^{N} \hat{A}^{n+1} * \left(1 - 2 * \hat{B}^{n+1}\right)\right]$$

We also count how many times the casino lets card counters play, as another performance metric.

Part III: Description of Policies

Policy Search: Boltzmann Exploration

$P(x^n = 1) = \frac{e^{\theta_B * \mu_1}}{e^{\theta_B * \mu_0} + e^{\theta_B * \mu_1}}$, where θ_B is a tunable parameter.

μ_1 is the reward (expected one period revenue) from banning the current player:
$\mu_1 = \mathbb{E}\left(\hat{A}^{n+1} * \left(1 - 2 * \hat{B}^{n+1}\right) \middle| \bar{a}^0, P_0(C = 1)\right)$, where $\bar{a}^0$ is average bet size from the prior casino record and $P_0(C = 1)$ is average chance of a card counter from the casino record.
$\mathbb{E}\left[\hat{A}^{n+1}\right] = \bar{a}^0$ (prior average bet size) because players are independent from each other.
Let $\bar{p}_1$ be $P(C = 1 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, \bar{a}^0) = \sum_{i=1}^{4} \mathrm{q}_\mathrm{i}^\mathrm{n} * P\left(C = 1 \mid \vec{\theta}^i, \bar{a}^0\right) = \sum_{i=1}^{4} \mathrm{q}_\mathrm{i}^\mathrm{n} * \min\left(< \vec{\theta}^i, \overrightarrow{\bar{a}^0} >^+, 1\right)$.
Essentially, $\bar{p}_1$ is a sum of the probabilities of card counting for each belief, weighted by the probabilities that we think belief i is the truth.

$$\mathbb{E}\left[\hat{B}^{n+1}\right] = P\left(\hat{A}^{n+1} = 1 \mid C = 1\right) * P(C = 1 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, a^0) + P\left(\hat{A}^{n+1} = 1 \mid C = 0\right) * P(C = 0 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, \bar{a}^0)$$
$$= .52\bar{p}_1 + .4(1 - \bar{p}_1)$$

Thus, $\mu_1 = \bar{a}^0(1 - 2(.52\bar{p}_1 + .4(1 - \bar{p}_1)))$

μ_0 is the reward from letting player continue, which we calculate in the same way as $\mu_1$but using $\bar{a}_t$ instead of a_0:
$\mathbb{E}\left[\hat{A}^{n+1}\right] = \bar{a}^n$.
Let $\bar{p}_0$ be $P(C = 1 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, \bar{a}_t) = \sum_{i=1}^{4} \mathrm{q}_\mathrm{i}^\mathrm{n} * P\left(C = 1 \mid \vec{\theta}^i, \bar{a}_t\right) = \sum_{i=1}^{4} \mathrm{q}_\mathrm{i}^\mathrm{n} * \min\left(< \vec{\theta}^i, \bar{a}_t >^+, 1\right)$.
Essentially, $\bar{p}_1$ is a sum of the probabilities of card counting for each belief, weighted by the probabilities that we think belief i is the truth.

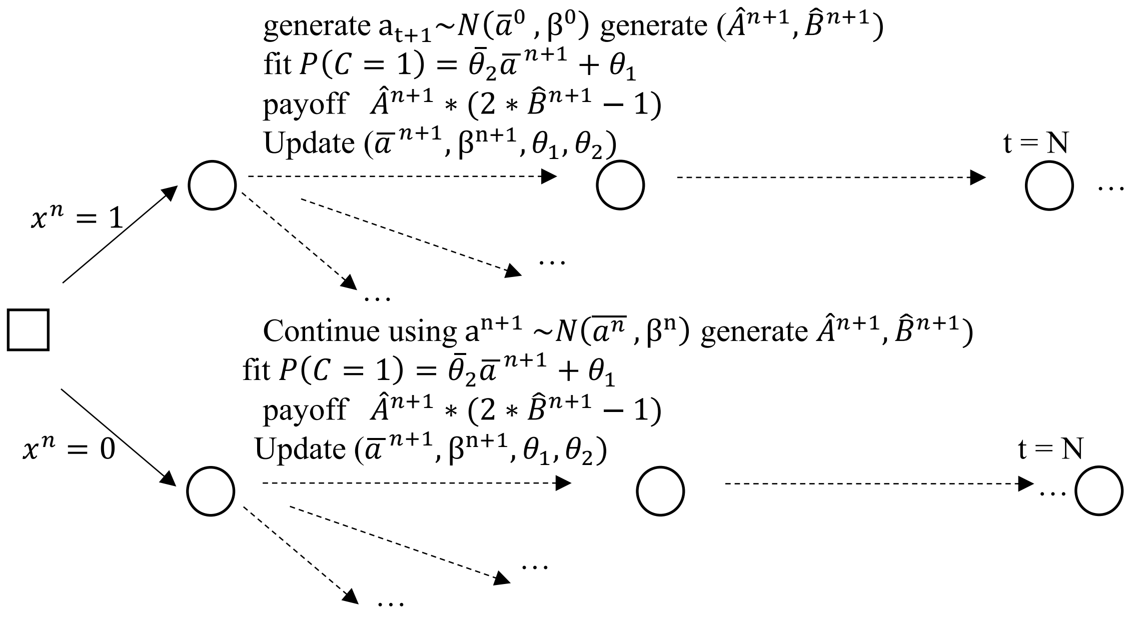
$$\mathbb{E}\left[\hat{B}^{n+1}\right] = P\left(\hat{A}^{n+1} = 1 \mid C = 1\right) * P(C = 1 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, \bar{a}^n\ ) + P\left(\hat{A}^{n+1} = 1 \mid C = 0\right) * P(C = 0 \mid \mathrm{q}_\mathrm{i}^\mathrm{n}, \bar{a}^n)$$
$$= .52\bar{p}_0 + .4(1 - \bar{p}_0)$$

Thus, $\mu_0 = P(C = 1 | S^{n+1}) * \mathbb{E}\left(\hat{A}_{t+1} * \left(1 - 2 * \hat{B}^{n+1}\right) \middle| C = 1\right) + P(C = 0 | S^{n+1}) * \mathbb{E}\left(\hat{A}^{n+1} * \left(1 - 2 * \hat{B}^{n+1}\right) \middle| C = 0\right)$
$= \bar{a}^n(1 - 2(.52\bar{p}_1 + .4(1 - \bar{p}_1)))$

Then we decide to ban the player with $P(x^n = 1)$, and let the player continue otherwise.

Look Ahead Policy: Tree Search

We evaluate the cumulative performance of the decisions to ban a player or let them continue at time n. That is, we look at the cumulative rewards from time n to N for both cases where we let the player continue compared to if we let ban them. We calculate the cumulative rewards by simulating many sample paths and taking their average.


generate $a_{t+1} \sim N(\bar{a}^0, \beta^0)$ generate $(\hat{A}^{n+1}, \hat{B}^{n+1})$
fit $P(C=1) = \bar{\theta}_2 \bar{a}^{n+1} + \theta_1$
payoff $\hat{A}^{n+1} * (2 * \hat{B}^{n+1} - 1)$
Update $(\bar{a}^{n+1}, \beta^{n+1}, \theta_1, \theta_2)$
t = N
$x^n = 1$
Continue using $a^{n+1} \sim N(\overline{a^n}, \beta^n)$ generate $\hat{A}^{n+1}, \hat{B}^{n+1})$
fit $P(C=1) = \bar{\theta}_2 \bar{a}^{n+1} + \theta_1$
payoff $\hat{A}^{n+1} * (2 * \hat{B}^{n+1} - 1)$
Update $(\bar{a}^{n+1}, \beta^{n+1}, \theta_1, \theta_2)$
t = N
$x^n = 0$

$$x^n = argmax_{\{0,1\}} E \sum_{t=n}^{N} C(S^n, X^\pi(S^n), W^{n+1}) = argmax_{\{0,1\}} \sum_{i=1}^{K} \sum_{t=n}^{N} C(S^n, X^\pi(S^n), W^{n+1})$$

Then, we compare expected cumulative rewards until time N of $x^n = 1$ with those of $x^n = 0$, and choose the decision with greater reward.

Part IV: Numerical Testing and Extension

We run 1000 trials of our simulation for N = 100 over the Boltzmann policy, tree search policy, and a pure exploration (random) policy and look at:

1. Cumulative revenue
2. Performance measured as the number of times the casino lets card counters play
3. Number of players banned

Boltzmann:

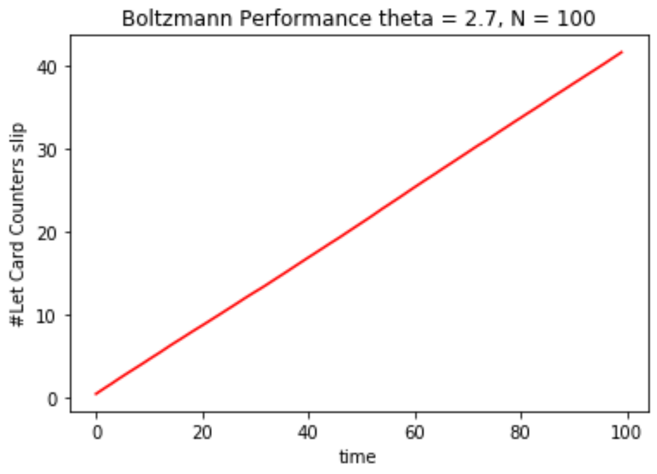
We tuned our Boltzmann parameter θ_B by finding the θ_B that maximizes the cumulative reward (revenue) for the Boltzmann policy.

$$\theta_B^* = argmax_{\theta_B} \mathbb{E}\left[\sum_{n=1}^{N} \hat{A}^{n+1} * \left(1 - 2 * \hat{B}^{n+1}\right)\right]$$

We tried θ_B values from .01 to 5, and found the optimal θ_B^* = 2.7. Some sample values of the cumulative revenue for different θ_B:

θ_B	cumulative revenue
0.1	5649.8
1	5813.62
1.5	5777.11
2.7	5919.59
3	5643.99
5	5592.02
10	5754.51

Over the 1000 trials, the tuned Boltzmann policy achieved 5919.59 cumulative revenue, 42619 rounds where card counters played, and 49817 bans.



Sample Boltzmann Performance Path

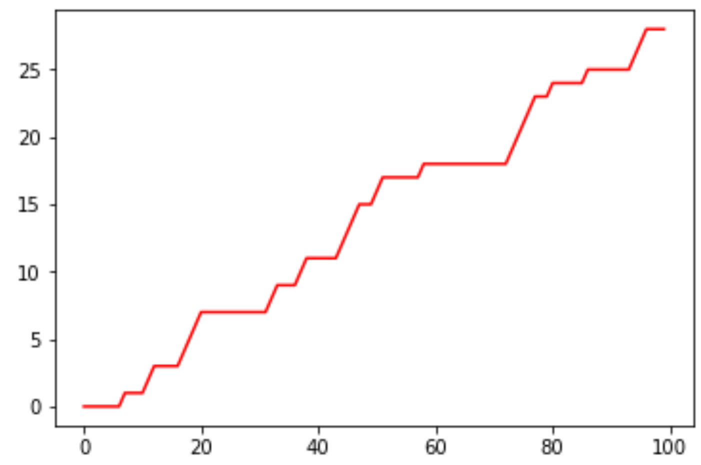
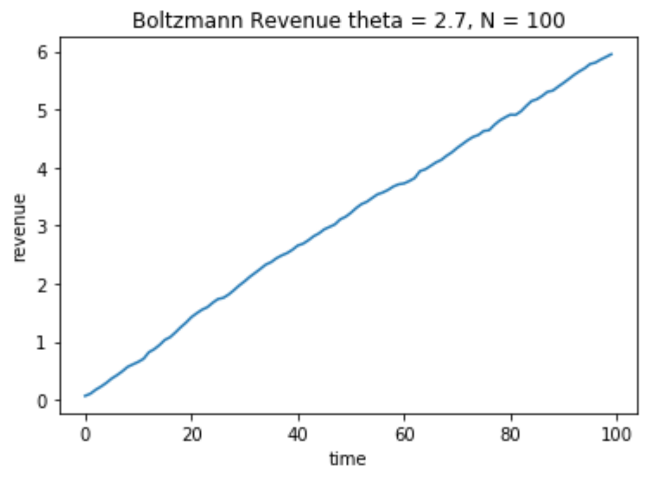


Fig. 1: Over 1000 trials and N = 100, the average performance (number of times casino lets card counters play), and a sample performance path under tuned $\theta_B = 2.7$. Note that cumulative performance is steadily increasing, while the sample performance path increases in steps, where the increases in the graph represent when a card counter is playing, and when the line stops increasing is when the casino has banned the card counter and a non card counter is playing.



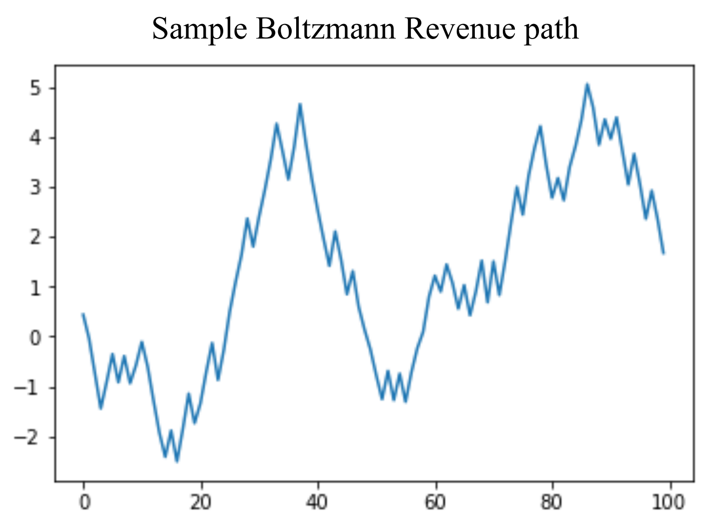
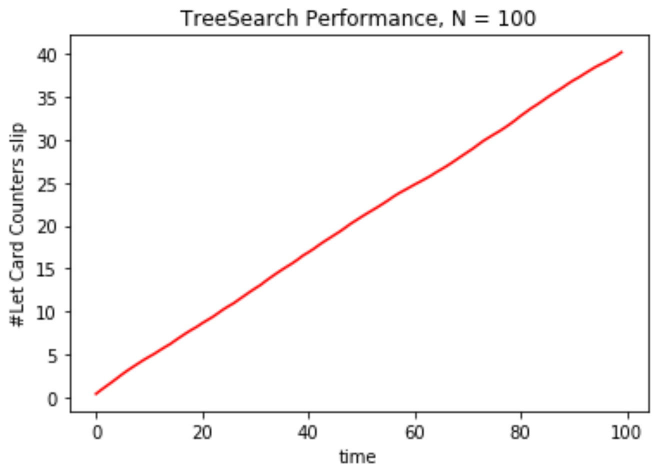


Fig. 2: Over 1000 trials and N = 100, the average cumulative revenue, and a sample revenue path under tuned Boltzmann. Note that cumulative revenue is steadily linearly increasing, while the individual revenue path has fluctuations representing wins and loses.

Tree Search:
Over the 1000 trials, the tree search policy achieved 5965.2 cumulative revenue, 40189 rounds where card counters played, and 45153 bans. Compared to the tuned Boltzmann policy, the tree search policy generated more cumulative revenue, let card counters play less rounds, and banned less people. It is interesting to note that the difference in these metrics between these two policies is not large, indicating that the tuned Boltzmann policy performs almost as well as the lookahead tree search policy.



Sample Tree Search Performance Path

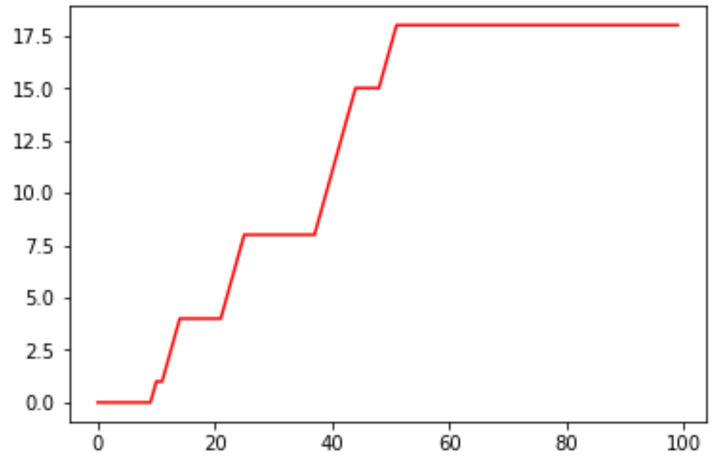
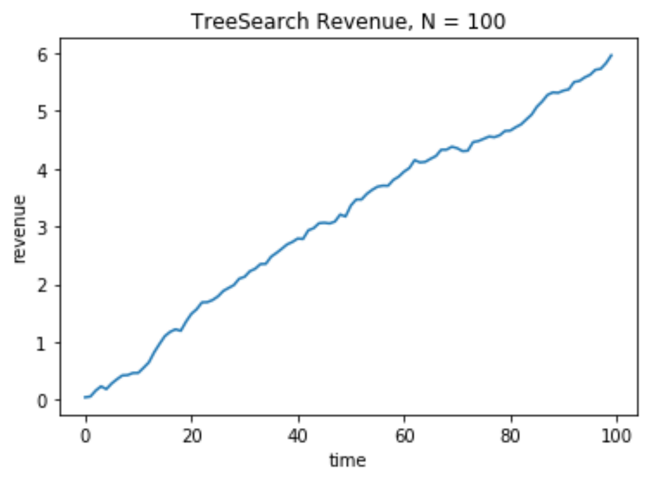


Fig. 3: Over 1000 trials and N = 100, the average performance (number of times casino lets card counters play), and a sample performance path under the tree search policy. Note that cumulative performance is steadily increasing, while the sample performance path increases in steps, where the increases in the graph represent when a card counter is playing, and when the line stops increasing is when the casino has banned the card counter and a non card counter is playing.



Sample Tree Search Revenue Path

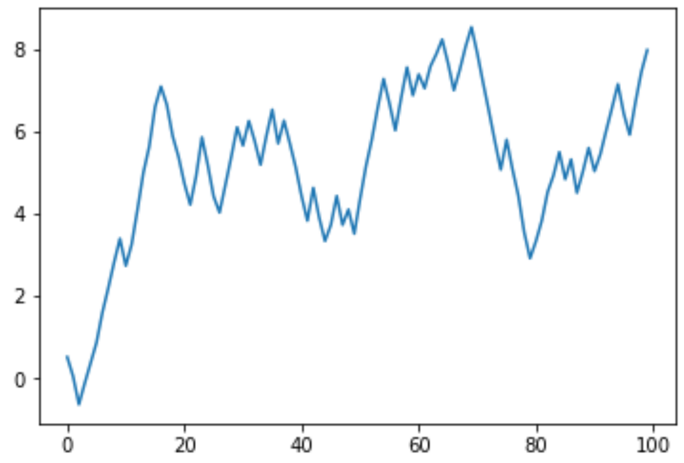
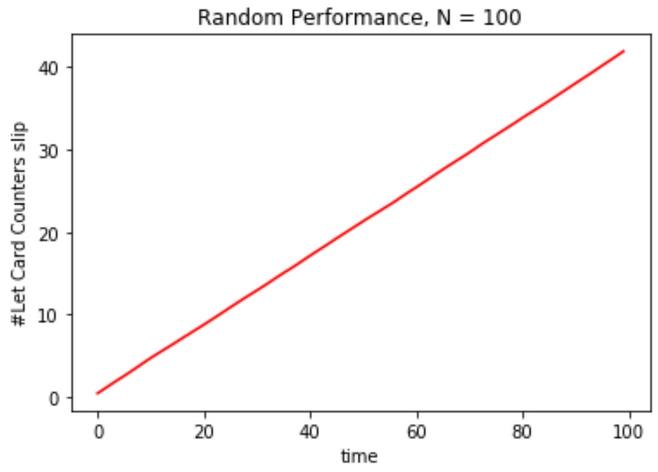


Fig. 4: Over 1000 trials and N = 100, the average cumulative revenue, and a sample performance path under the tree search policy. Note that cumulative revenue is steadily linearly increasing, while the individual revenue path has fluctuations representing wins and loses.

Pure Exploration (Random):
For the pure exploration policy, we randomly ban players at each time with half probability. Overall, both other policies outperformed the pure exploration policy, which is to be expected because pure exploration policies do not work very well for online learning problems. The policy gave 5531.78 cumulative revenue, 41927 rounds where card counters played, and 50036 bans.



Sample Random Performance Path

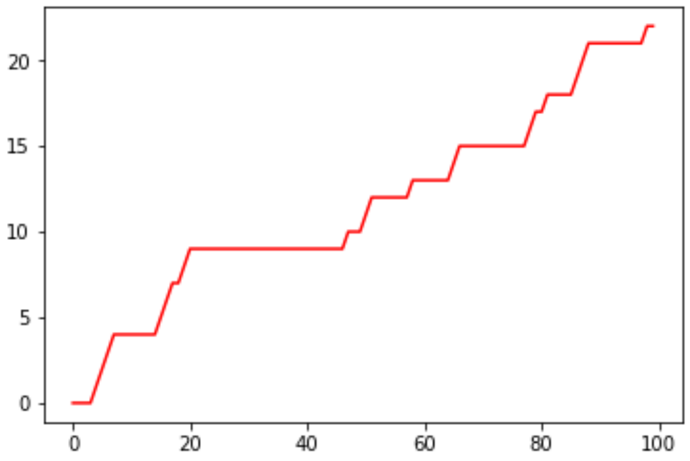


Fig. 5: Over 1000 trials and N = 100, the average performance (number of times casino lets card counters play), and a sample performance path under the pure exploration policy. Note that cumulative performance is steadily increasing, while the sample performance path increases in steps, where the increases in the graph represent when a card counter is playing, and when the line stops increasing is when the casino has banned the card counter and a non card counter is playing.

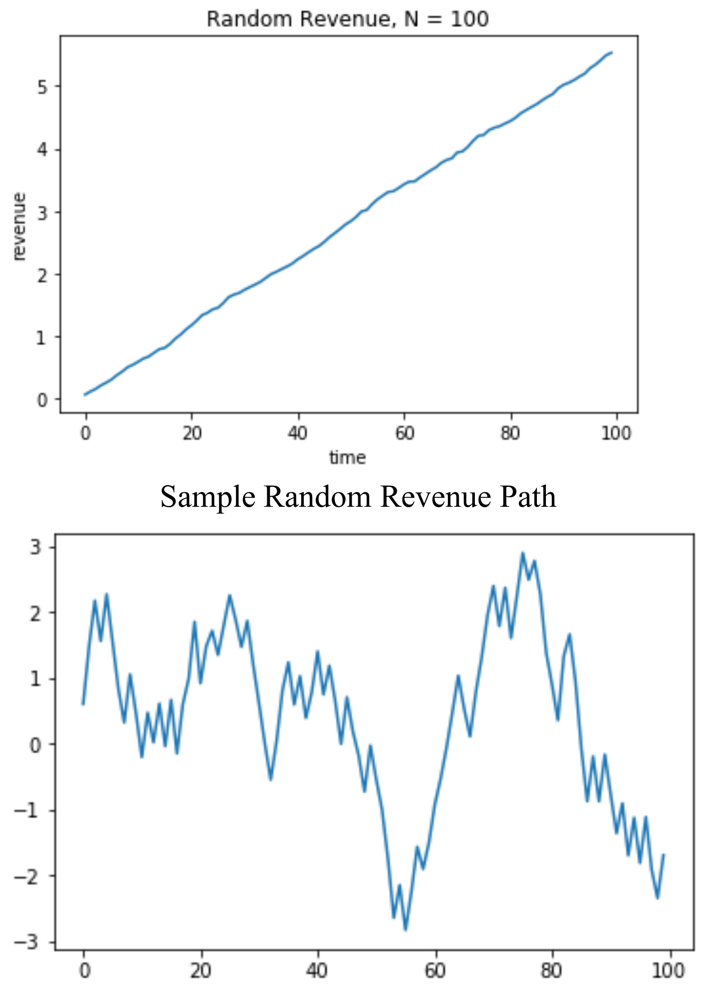
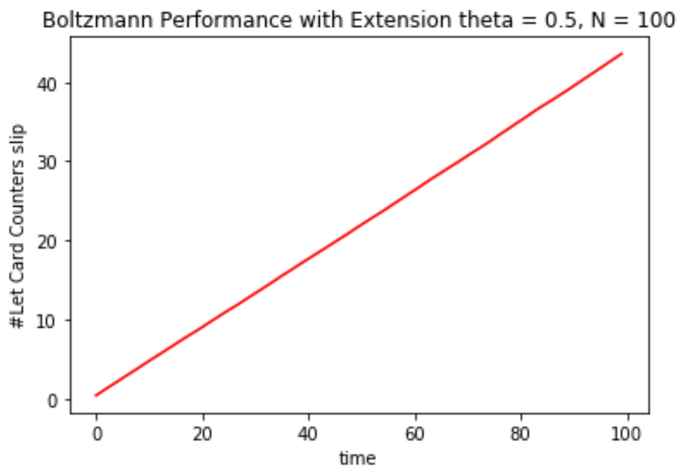


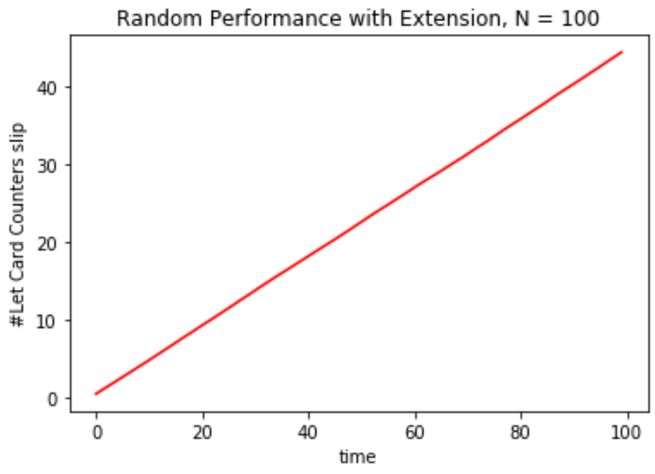
Fig. 6: Over 1000 trials and N = 100, the average cumulative revenue, and a sample performance path under the pure exploration policy. Note that cumulative revenue is steadily linearly increasing, while the individual revenue path has fluctuations representing wins and loses.

Model extension:
We can modify our belief model by letting true $\bar{a}$ that each player sampled from be a random variable, compared to a constant, deterministic $\bar{a}$ in our previous model. When the jth new player initially comes to the table, we generate new truth $\tilde{a}^j \sim N(\tilde{a}^{j-1}, \beta^a)$ and set $\bar{a} = \tilde{a}^j$ for the jth new player. In our simulation, we set $\beta^a = .05$. That is, $\tilde{a}^j$ is a random walk starting at $\tilde{a}^0 = \bar{a}$.

We retuned the Boltzmann policy with the extension, and $\theta_B^* = .5$ gave us 5204.35 cumulative revenue, 43195 rounds where card counters played, and 49934 bans. The tree search policy gave us 5491.67 cumulative revenue, 39252 rounds where card counters played, and 50958 bans. The pure exploration policy gave us 4753.05 cumulative revenue, 44311 rounds where card counters played, and 49921 bans.

The overall rankings stayed the same in terms of the tree search policy performing better than Boltzmann and both better than the pure exploration. However, as to be expected, the policies performed worse in the extension compared to the original model.





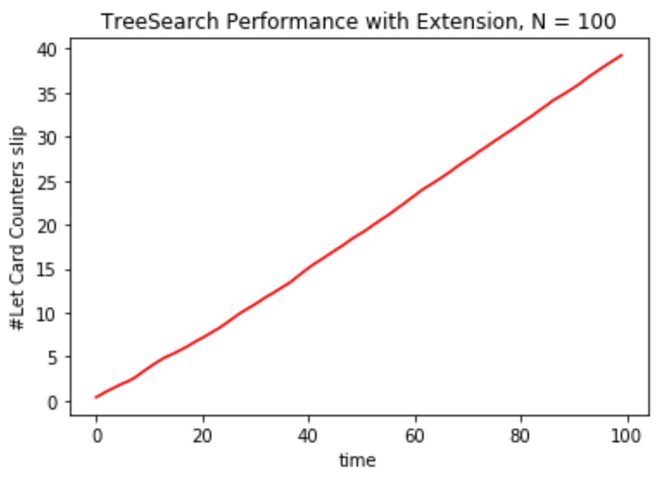
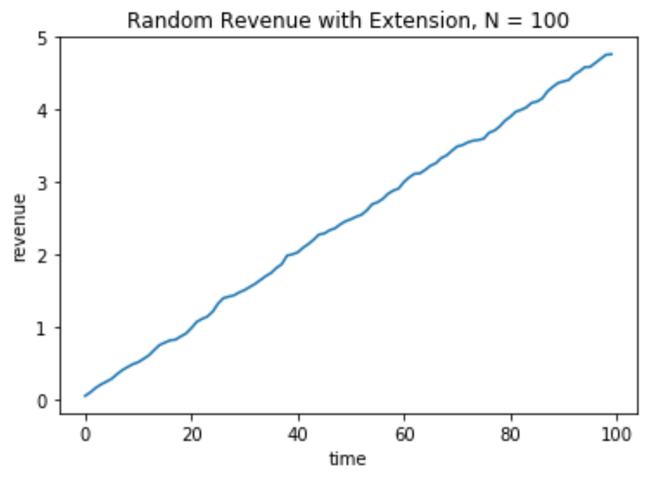


Fig. 7: Performance of the Boltzmann policy, tree search policy, and exploration policy with the model extension. Note that compared to the respective performances of the policies before the extension, these policies with the extension perform worse (card counters play more rounds)



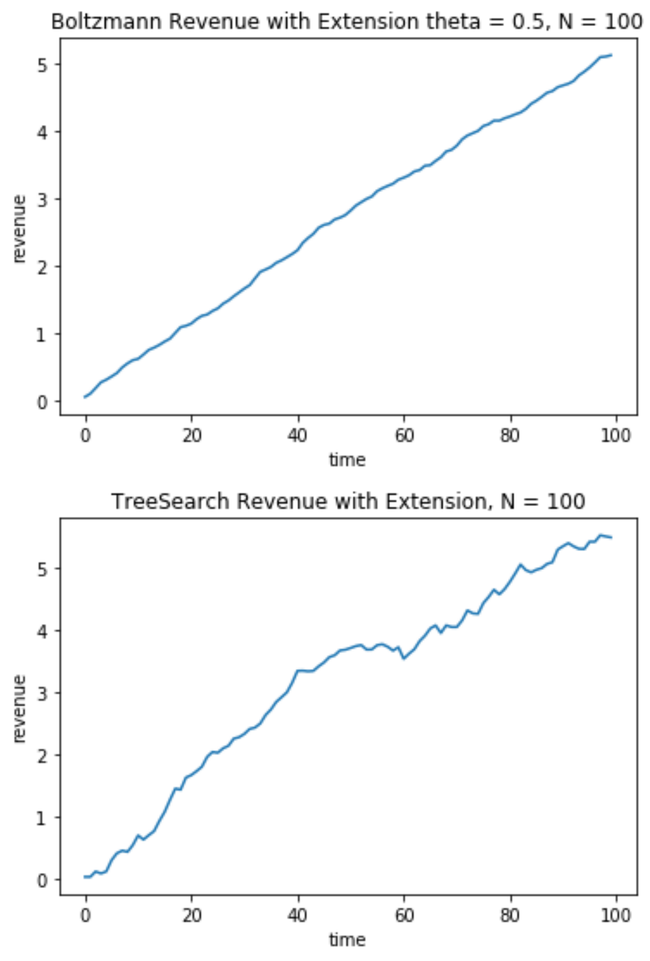


Fig. 8: Average revenue paths of the Boltzmann policy, tree search policy, and exploration policy with the model extension. While revenue still increases on the whole, note that compared to the respective revenues of the policies before the extension, these policies with the extension perform worse.

Part V: Appendix: Code

```
# -*- coding: utf-8 -*-
"""
Created on Tue May  1 16:02:54 2018

@author: Woramanot and Connie Zhu
"""

# In[]:

import numpy as np
from matplotlib.pyplot import *

# Set of possible true parametric curve
theta1 = [0,-0.5,0.5,0.5]
theta2 = [0.5,1,0.5,0]

#theta1 = [0,0,0,0]

#theta2 = [0,0,0.5,0.5]

# True parameter
a_bar = 0.6

# Write the function to test the policy

def TestPolicy(policy = 0, N = 100, Tuner = 1, Extension = 0):
    # First, define the truth
    truth = np.random.randint(0,4)
    true_theta1 = theta1[truth]
    true_theta2 = theta2[truth]

    # Set precision Variance =0.01
    Beta_W = 100
    # Truth
    a_bar = 0.6
    # Casino record
    a_0 = 0.5
    Beta_0 = 100
    # Uniform prior
    q_vector = [.25]*4

    # Create empty variable for storing objective function
    # Check performance too
    revenue = 0
```

```
revenue_vector = [0]*N
Performance = 0
Performance_vector = [0]*N
# numbers of kick made
Kick = 0
# Make new player ticker
NewPlayer = 1
#Iterate through n periods
for t in range(N):
    # Generate truth for a new player
    if NewPlayer == 1:
        NewPlayer = 0
        # Casino's belief is renewed as new player comes in
        Beta_t = Beta_0
        a_t = a_0
        #Extension is changing truth random-walk like for each new player
        if Extension == 1:
            a_bar = np.random.normal(a_bar,0.05)
        # Truth
        a = np.random.normal(a_bar,0.1)
        # Generate if card counter or not
        temp = np.random.random()
        if temp <= a*true_theta2 + true_theta1:
            C = 1
        else:
            C = 0
        # They then play a game
        # generate average bet size
        A_t = np.random.normal(a,0.1)
        # Win or not
        if C == 1:
            B_t = np.random.random()
            if B_t <= 0.52:
                B_t = 1
            else:
                B_t = 0
        if C== 0:
            B_t = np.random.random()
            if B_t <= 0.4:
                B_t = 1
            else:
                B_t = 0
    if NewPlayer == 0:
        # They then play a game
        # generate average bet size
        A_t = np.random.normal(a,0.1)
    # Win or not
    if C == 1:
        B_t = np.random.random()
        if B_t <= 0.52:
            B_t = 1
        else:
            B_t = 0
    if C== 0:
        B_t = np.random.random()
        if B_t <= 0.4:
            B_t = 1
        else:
            B_t = 0
# Bayesian Updating
# Transition function
a_t = np.divide(Beta_t*a_t + Beta_W*A_t,Beta_t + Beta_W)
Beta_t = Beta_t + Beta_W

# Update q depending on B_t
if B_t == 1:
    temp = [0]*4
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_t
        prob = min(prob, 1)
        prob = max(prob, 0)
        temp[r] = q_vector[r]*(.52*prob + .4*(1 - prob))
    total = sum(temp)
    temp = temp/total
    q_vector = temp
else:
    temp = [0]*4
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_t
        prob = min(prob, 1)
        prob = max(prob, 0)
        temp[r] = q_vector[r]*(.48*prob + .6*(1 - prob))
    total = sum(temp)
    temp = temp/total
    q_vector = temp

# Update revenue for period t
revenue = revenue + A_t*(1-2*B_t)
revenue_vector[t] = revenue
if C == 1:
    Performance = Performance + 1
Performance_vector[t] = Performance
```

```
# Decision Policies

if policy == 0:
    ###### Boltzmann Policy
    # Calculate mu_1 (if kick) and mu_0 (if not)
    # Find weighted sum of P(C=1)
    weighted_sum = 0
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_0
        prob = min(prob,1)
        prob = max(prob,0)
        weighted_sum = weighted_sum + q_vector[r]*prob
    Expected_B = 0.52*weighted_sum + 0.4*(1-weighted_sum)
    mu_1 = a_0*(1 - 2*Expected_B)

    weighted_sum = 0
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_t
        prob = min(prob,1)
        prob = max(prob,0)
        weighted_sum = weighted_sum + q_vector[r]*prob
    Expected_B = 0.52*weighted_sum + 0.4*(1-weighted_sum)
    mu_0 = a_t*(1 - 2*Expected_B)

    # Create a probability for banning
    Prob_Ban = np.divide(np.exp(Tuner*mu_1),np.exp(Tuner*mu_1)+np.exp(Tuner*mu_0))
    temp = np.random.random()
    if temp <= Prob_Ban:
        NewPlayer = 1
        Kick = Kick + 1
if policy == 1:
    ##### Tree Search Policy
    # Calculate mu_1 (if kick) and mu_0 (if not)
    mu_1 = 0
    mu_0 = 0
    # P(C=1) if kick
    PC1_1 = 0
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_0
        prob = min(prob,1)
        prob = max(prob,0)
        PC1_1 = PC1_1 + q_vector[r]*prob
    # P(C=1) if not kick
    PC1_0 = 0
    for r in range(4):
        prob = theta1[r] + theta2[r]*a_t
        prob = min(prob,1)
        prob = max(prob,0)
        PC1_0 = PC1_0 + q_vector[r]*prob
    # Loop #100 times to update mu_1 and mu_0
    for r in range(100):
        # Update mu_1 if kick out
        a_temp = a_0
        Beta_temp = Beta_0
        # If kick out
        # Generate if card counter or not
        temp = np.random.random()
        if temp <= PC1_1:
            C_temp = 1
        else:
            C_temp = 0
        # Simulate until endtime
        for i in range(N-t):
            # They then play a game
            # generate average bet size
            A_temp = np.random.normal(a_temp,0.1)
            # Win or not
            if C_temp == 1:
                B_temp = np.random.random()
                if B_temp <= 0.52:
                    B_temp = 1
                else:
                    B_temp = 0
            if C_temp== 0:
                B_temp = np.random.random()
                if B_temp <= 0.4:
                    B_temp = 1
                else:
                    B_temp = 0
            # Transition function
            a_temp = np.divide(Beta_temp*a_temp + Beta_W*A_temp,Beta_temp + Beta_W)
            Beta_temp = Beta_temp + Beta_W
            # Update mu_1
            mu_1 = mu_1 + A_temp*(1-2*B_temp)

        # Update mu_0 if let the player continues
        a_temp = a_t
        Beta_temp = Beta_t
        # If kick out
        # Generate if card counter or not
        temp = np.random.random()
        if temp <= PC1_0:
                C_temp = 1
            else:
                C_temp = 0
            # Simulate until endtime
            for i in range(N-t):
                # They then play a game
                # generate average bet size
                A_temp = np.random.normal(a_temp,0.1)
                # Win or not
                if C_temp == 1:
                    B_temp = np.random.random()
                    if B_temp <= 0.52:
                        B_temp = 1
                    else:
                        B_temp = 0
                if C_temp== 0:
                    B_temp = np.random.random()
                    if B_temp <= 0.4:
                        B_temp = 1
                    else:
                        B_temp = 0
                # Transition function
                a_temp = np.divide(Beta_temp*a_temp + Beta_W*A_temp,Beta_temp + Beta_W)
                Beta_temp = Beta_temp + Beta_W
                # Update mu_0
                mu_0 = mu_0 + A_temp*(1-2*B_temp)
        # Kick a player if the expected gain from new player is better
        if mu_1 >= mu_0:
            NewPlayer = 1
            Kick = Kick + 1

    return revenue_vector, Performance_vector, revenue, Performance, q_vector,truth, Kick

# In[]:

revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 1)

plot(Performance_vector)
print(revenue)
print(Performance)
print(q_vector)
print(truth)


# In[]:
```

```
revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100)

plot(Performance_vector)
print(revenue)
print(Performance)
print(q_vector)
print(truth)

# In[]:
# Now we compare performance of Boltzmann with Tree Search
Random = [0,0,0]
Boltzmann_1 = [0,0,0]
Boltzmann_01 = [0,0,0]
Boltzmann_10 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0)
    Random[0] = Random[0] + Performance
    Random[1] = Random[1] + revenue
    Random[2] = Random[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1)
    Boltzmann_1[0] = Boltzmann_1[0] + Performance
    Boltzmann_1[1] = Boltzmann_1[1] + revenue
    Boltzmann_1[2] = Boltzmann_1[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.1)
    Boltzmann_01[0] = Boltzmann_01[0] + Performance
    Boltzmann_01[1] = Boltzmann_01[1] + revenue
    Boltzmann_01[2] = Boltzmann_01[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 10)
    Boltzmann_10[0] = Boltzmann_10[0] + Performance
    Boltzmann_10[1] = Boltzmann_10[1] + revenue
    Boltzmann_10[2] = Boltzmann_10[2] + Kick
print(Random)
print(Boltzmann_1)
print(Boltzmann_01)
print(Boltzmann_10)
#[41674, 5408.117902419714, 50077]
#[42626, 5813.620664030448, 49847]
#[41754, 5649.790334589908, 50145]
#[42712, 5754.505910242727, 49625]
# In[]:
# Now we optimize Boltzmann
```

```
Boltzmann_5 = [0,0,0]
Boltzmann_1 = [0,0,0]
Boltzmann_05 = [0,0,0]
Boltzmann_15 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 5)
    Boltzmann_5[0] = Boltzmann_5[0] + Performance
    Boltzmann_5[1] = Boltzmann_5[1] + revenue
    Boltzmann_5[2] = Boltzmann_5[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1)
    Boltzmann_1[0] = Boltzmann_1[0] + Performance
    Boltzmann_1[1] = Boltzmann_1[1] + revenue
    Boltzmann_1[2] = Boltzmann_1[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.5)
    Boltzmann_05[0] = Boltzmann_05[0] + Performance
    Boltzmann_05[1] = Boltzmann_05[1] + revenue
    Boltzmann_05[2] = Boltzmann_05[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1.5)
    Boltzmann_15[0] = Boltzmann_15[0] + Performance
    Boltzmann_15[1] = Boltzmann_15[1] + revenue
    Boltzmann_15[2] = Boltzmann_15[2] + Kick
print(Boltzmann_5)
print(Boltzmann_1)
print(Boltzmann_05)
print(Boltzmann_15)

# In[]:
# Now we optimize Boltzmann
Boltzmann_42 = [0,0,0]
Boltzmann_3 = [0,0,0]
Boltzmann_27 = [0,0,0]
Boltzmann_15 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 4.2)
    Boltzmann_42[0] = Boltzmann_42[0] + Performance
    Boltzmann_42[1] = Boltzmann_42[1] + revenue
    Boltzmann_42[2] = Boltzmann_42[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 3)
    Boltzmann_3[0] = Boltzmann_3[0] + Performance
    Boltzmann_3[1] = Boltzmann_3[1] + revenue
    Boltzmann_3[2] = Boltzmann_3[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 2.7)
    Boltzmann_27[0] = Boltzmann_27[0] + Performance
    Boltzmann_27[1] = Boltzmann_27[1] + revenue
    Boltzmann_27[2] = Boltzmann_27[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1.5)
    Boltzmann_15[0] = Boltzmann_15[0] + Performance
    Boltzmann_15[1] = Boltzmann_15[1] + revenue
    Boltzmann_15[2] = Boltzmann_15[2] + Kick
print(Boltzmann_42)
print(Boltzmann_3)
print(Boltzmann_27)
print(Boltzmann_15)
#[42579, 5681.238704351455, 49665]
#[42488, 5643.987752426847, 49994]
#[42619, 5919.59100104878, 49817]
#[43482, 5775.342417041388, 49829]
# Theta = 2.7 is optimal
# In[]:
# Now we get performance of Tree Search
# And Draw a graph
TreeSearch = [0,0,0]
TreeSearch_G = np.zeros([2,100])
for i in range(100):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 1, N = 100)
    TreeSearch[0] = TreeSearch[0] + Performance
    TreeSearch[1] = TreeSearch[1] + revenue
    TreeSearch[2] = TreeSearch[2] + Kick
    TreeSearch_G[0,:] = TreeSearch_G[0,:] + revenue_vector
    TreeSearch_G[1,:] = TreeSearch_G[1,:] + Performance_vector

# In[]:

figure(0)
plot(np.divide(TreeSearch_G[0,:],100))
title('TreeSearch Revenue, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(TreeSearch_G[1,:],100), 'r')
title('TreeSearch Performance, N = 100')
xlabel('time')
ylabel('#Let Card Counters slip')
```

```
# In[]:
# Now we draw Boltzmann Graph
Boltzmann_27G = np.zeros([2,100])
Boltzmann_27 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 2.7)
    Boltzmann_27[0] = Boltzmann_27[0] + Performance
    Boltzmann_27[1] = Boltzmann_27[1] + revenue
    Boltzmann_27[2] = Boltzmann_27[2] + Kick
    Boltzmann_27G[0,:] = Boltzmann_27G[0,:] + revenue_vector
    Boltzmann_27G[1,:] = Boltzmann_27G[1,:] + Performance_vector

# In[]:
figure(0)
plot(np.divide(Boltzmann_27G[0,:],1000))
title('Boltzmann Revenue theta = 2.7, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(Boltzmann_27G[1,:],1000), 'r')
title('Boltzmann Performance theta = 2.7, N = 100')
xlabel('time')
ylabel('#Let Card Counters slip')
# In[]:
# Now we draw Random
RandomG = np.zeros([2,100])
Random = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0)
    Random[0] = Random[0] + Performance
    Random[1] = Random[1] + revenue
    Random[2] = Random[2] + Kick
    RandomG[0,:] = RandomG[0,:] + revenue_vector
    RandomG[1,:] = RandomG[1,:] + Performance_vector
# In[]:
figure(0)
plot(np.divide(RandomG[0,:],1000))
title('Random Revenue, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(RandomG[1,:],1000), 'r')
title('Random Performance, N = 100')
xlabel('time')
```

```
ylabel('#Let Card Counters slip')
# In[]:

# Restart the whole process now with Extension
# In[]:
# Now we compare performance of Boltzmann with Tree Search
Random = [0,0,0]
Boltzmann_1 = [0,0,0]
Boltzmann_01 = [0,0,0]
Boltzmann_10 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0, Extension = 1)
    Random[0] = Random[0] + Performance
    Random[1] = Random[1] + revenue
    Random[2] = Random[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1, Extension = 1)
    Boltzmann_1[0] = Boltzmann_1[0] + Performance
    Boltzmann_1[1] = Boltzmann_1[1] + revenue
    Boltzmann_1[2] = Boltzmann_1[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.1, Extension = 1)
    Boltzmann_01[0] = Boltzmann_01[0] + Performance
    Boltzmann_01[1] = Boltzmann_01[1] + revenue
    Boltzmann_01[2] = Boltzmann_01[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 10, Extension = 1)
    Boltzmann_10[0] = Boltzmann_10[0] + Performance
    Boltzmann_10[1] = Boltzmann_10[1] + revenue
    Boltzmann_10[2] = Boltzmann_10[2] + Kick
print(Random)
print(Boltzmann_1)
print(Boltzmann_01)
print(Boltzmann_10)
# In[]:
# Now we optimize Boltzmann
Boltzmann_5 = [0,0,0]
Boltzmann_1 = [0,0,0]
Boltzmann_05 = [0,0,0]
Boltzmann_15 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 5, Extension = 1)
    Boltzmann_5[0] = Boltzmann_5[0] + Performance
    Boltzmann_5[1] = Boltzmann_5[1] + revenue
    Boltzmann_5[2] = Boltzmann_5[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1, Extension = 1)
    Boltzmann_1[0] = Boltzmann_1[0] + Performance
    Boltzmann_1[1] = Boltzmann_1[1] + revenue
    Boltzmann_1[2] = Boltzmann_1[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.5, Extension = 1)
    Boltzmann_05[0] = Boltzmann_05[0] + Performance
    Boltzmann_05[1] = Boltzmann_05[1] + revenue
    Boltzmann_05[2] = Boltzmann_05[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 1.5, Extension = 1)
    Boltzmann_15[0] = Boltzmann_15[0] + Performance
    Boltzmann_15[1] = Boltzmann_15[1] + revenue
    Boltzmann_15[2] = Boltzmann_15[2] + Kick
print(Boltzmann_5)
print(Boltzmann_1)
print(Boltzmann_05)
print(Boltzmann_15)
# In[]:
# Now we optimize Boltzmann
Boltzmann_05 = [0,0,0]
Boltzmann_033 = [0,0,0]
Boltzmann_08 = [0,0,0]
Boltzmann_066 = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.5, Extension = 1)
    Boltzmann_05[0] = Boltzmann_05[0] + Performance
    Boltzmann_05[1] = Boltzmann_05[1] + revenue
    Boltzmann_05[2] = Boltzmann_05[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.33, Extension = 1)
    Boltzmann_033[0] = Boltzmann_033[0] + Performance
    Boltzmann_033[1] = Boltzmann_033[1] + revenue
    Boltzmann_033[2] = Boltzmann_033[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.8, Extension = 1)
    Boltzmann_08[0] = Boltzmann_08[0] + Performance
    Boltzmann_08[1] = Boltzmann_08[1] + revenue
    Boltzmann_08[2] = Boltzmann_08[2] + Kick
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.66, Extension = 1)
    Boltzmann_066[0] = Boltzmann_066[0] + Performance
    Boltzmann_066[1] = Boltzmann_066[1] + revenue
    Boltzmann_066[2] = Boltzmann_066[2] + Kick
print(Boltzmann_05)
print(Boltzmann_033)
print(Boltzmann_08)
print(Boltzmann_066)

# In[]:

    # Draw Graphs for extension

# Now we get performance of Tree Search
TreeSearch_E = [0,0,0]
TreeSearch_G_E = np.zeros([2,100])
for i in range(100):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 1, N = 100, Extension = 1)
    TreeSearch_E[0] = TreeSearch_E[0] + Performance
    TreeSearch_E[1] = TreeSearch_E[1] + revenue
    TreeSearch_E[2] = TreeSearch_E[2] + Kick
    TreeSearch_G_E[0,:] = TreeSearch_G_E[0,:] + revenue_vector
    TreeSearch_G_E[1,:] = TreeSearch_G_E[1,:] + Performance_vector

# In[]:

figure(0)
plot(np.divide(TreeSearch_G_E[0,:],100))
title('TreeSearch Revenue with Extension, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(TreeSearch_G_E[1,:],100), 'r')
title('TreeSearch Performance with Extension, N = 100')
xlabel('time')
ylabel('#Let Card Counters slip')
# In[]:
# Now we draw Boltzmann Graph
Boltzmann_05G_E = np.zeros([2,100])
Boltzmann_05_E = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0.5, Extension = 1)
    Boltzmann_05_E[0] = Boltzmann_05_E[0] + Performance
    Boltzmann_05_E[1] = Boltzmann_05_E[1] + revenue
    Boltzmann_05_E[2] = Boltzmann_05_E[2] + Kick
    Boltzmann_05G_E[0,:] = Boltzmann_05G_E[0,:] + revenue_vector
    Boltzmann_05G_E[1,:] = Boltzmann_05G_E[1,:] + Performance_vector
```

```
# In[]:
figure(0)
plot(np.divide(Boltzmann_05G_E[0,:],1000))
title('Boltzmann Revenue with Extension theta = 0.5, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(Boltzmann_05G_E[1,:],1000), 'r')
title('Boltzmann Performance with Extension theta = 0.5, N = 100')
xlabel('time')
ylabel('#Let Card Counters slip')
# In[]:
# Now we draw Random with Extension
RandomG_E = np.zeros([2,100])
Random_E = [0,0,0]
for i in range(1000):
    revenue_vector, Performance_vector, revenue, Performance, q_vector,truth,Kick =
TestPolicy(policy = 0, N = 100, Tuner = 0, Extension = 1)
    Random_E[0] = Random_E[0] + Performance
    Random_E[1] = Random_E[1] + revenue
    Random_E[2] = Random_E[2] + Kick
    RandomG_E[0,:] = RandomG_E[0,:] + revenue_vector
    RandomG_E[1,:] = RandomG_E[1,:] + Performance_vector

# In[]:
figure(0)
plot(np.divide(RandomG_E[0,:],1000))
title('Random Revenue with Extension, N = 100')
xlabel('time')
ylabel('revenue')
figure(1)
plot(np.divide(RandomG_E[1,:],1000), 'r')
title('Random Performance with Extension, N = 100')
xlabel('time')
ylabel('#Let Card Counters slip')
```