Thompson Sampling on Poisson trails and global maximum of non-concave function

December 2023

1 Introduction

This project delves into the application of Thompson Sampling in two distinct areas. The primary focus is on optimizing predictive accuracy and maximizing payouts in scenarios involving Poisson trials of Bernoulli random variables, such as coin tosses. Additionally, the project explores the application of Thompson Sampling for the challenging task of finding the global maximum of non-concave functions.

2 Part 1

2.1 Problem Formulation

In this scenario, an enigmatic coin with an unknown probability of landing heads is repeatedly tossed. At each instance t, a player is tasked with predicting the outcome. If the player accurately guesses the outcome, they receive a payout of 1; otherwise, the payout is 0. The overarching goal is to devise a strategy that progressively maximizes the cumulative payout over time.

2.2 Algorithm

We begin with 2 arms, with Arm 1 representing the prediction of the outcome as heads, and Arm 0 representing the prediction of the outcome as tails. The parameters at any given time t are defined as follows:

- $N_k(t)$: The number of times the player has chosen Arm k up to time t.
- $Q_k(t)$: The mean payout associated with the chosen action of Arm k up to time t.

This structure allows us to keep track of the player's choices and the corresponding mean payouts over time.

Algorithm 1: Thompson Sampling of Coin Tosses

Initialization $Q_1 = Q_0 = 0$, $N_1 = N_0 = 0$, reward $= 0$, wealth $= 0$
for $t < N$ **do**
  Toss 2 coins $S_k = \text{Bern}(Q_k)$
  Compare the outcome of coin tosses $c = \arg\max(s_1, s_2)$
  Reward reward $= 1$ if S_c matches the real outcome, 0 otherwise.
  Update Q_c *and* N_c $N_c(t) = N_c(t-1) + 1$,
   $Q_c = \frac{N_c(t-1)Q_c(t-1)+\text{reward}}{N_c(t)}$
  Wealth wealth = wealth + reward

where N is the upper bound of experiment steps.

A suitable test scenario for evaluating the Thompson Sampling algorithm is to apply it to a biased coin that consistently lands heads. In this case, the algorithm's performance can be assessed by comparing the achieved wealth to the total number of iterations, N. Given that heads is always the correct outcome, a successful algorithm should yield a wealth close to N.

2.3 Result

The results are presented in the form of a table for $N = 100000$ across three cases, as well as a control case where:

- $P_0(H|t) = 1$
- $P_1(H|t) = \frac{3}{5}$
- $P_2(H|t) = t \mod 2$
- $P_3(H|t) = \lfloor \frac{\log t}{\log 2} \rfloor \mod 2$

Coin with probability	Q_1	Q_0	N_1 4	N_0	wealth
P_0	0.0	1.0	1	99999	99999
P_1	0.5984	0.4015	59691	40309	51956
P_2	0.4997	0.4997	50101	49899	49967
P_3	0.4523	0.5799	52166	47834	51334

The following figures depict the Equity curves and the Learning curves.

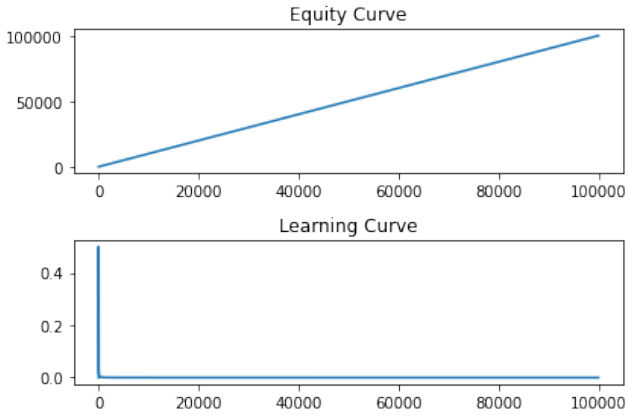

Figure 1: Equity curves and the Learning curves of Coin tosses with $P_0(H|t) = 1$

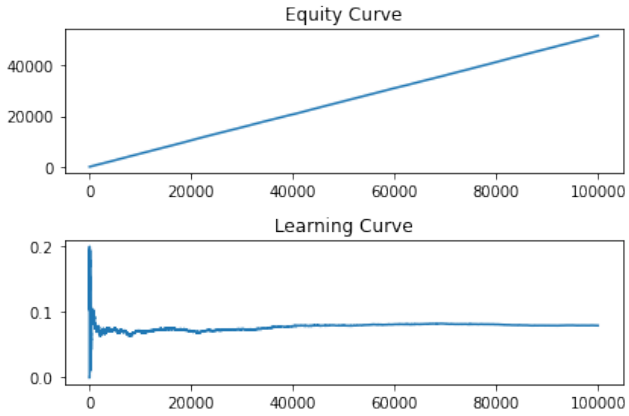

Figure 2: Equity curves and the Learning curves of Coin tosses with $P_1(H|t) = \frac{3}{5}$

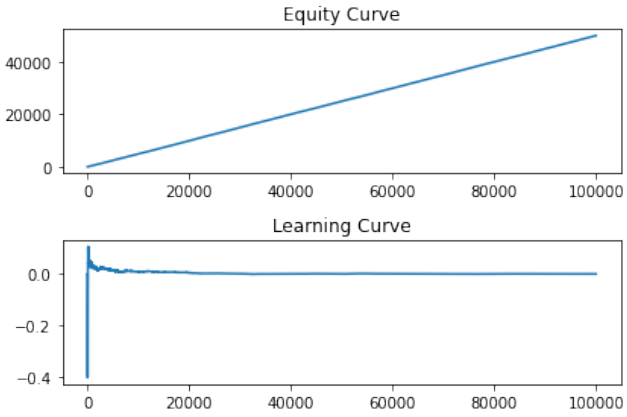

Figure 3: Equity curves and the Learning curves of Coin tosses with $P_2(H|t) = t \mod 2$

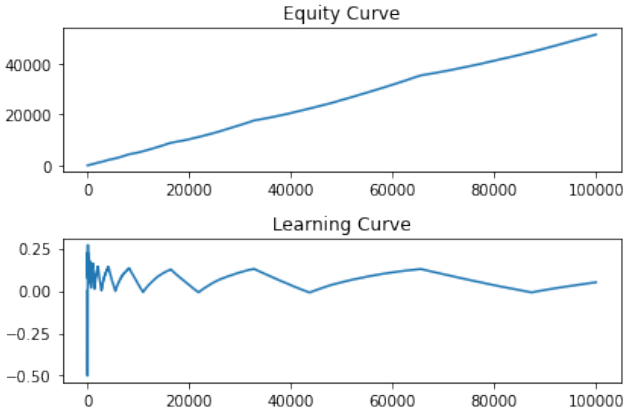

Figure 4: Equity curves and the Learning curves of Coin tosses with $P_3(H|t) = \lfloor \frac{\log t}{\log 2} \rfloor \mod 2$

The learning curve is obtained by plotting the average difference between the maximal wealth achievable by consistently choosing heads or tails and the evolving wealth over time t. Notably, when $P_1(H|t) = \frac{3}{5}$, the learning curve does not converge to 0. In an extreme scenario represented by $P_3(H|t) = \lfloor \frac{\log t}{\log 2} \rfloor \mod 2$, the learning curve exhibits oscillatory behavior.

3 Part 2

3.1 Problem Formulation

Part 2 involves the pursuit of the global maximum of a non-concave function within the interval $[0, 1]$. In contrast to concave functions, where global maxima can be readily obtained through standard ascending methods, non-concave functions necessitate estimation techniques to identify potential global maxima.

3.2 Algorithm

We first divide the unit interval into m equal-length sub intervals with partition

$$x_0 = 0 < x_1 < ... < x_m = 1$$

creating a mixture distribution of m uniform distributions on subintervals with proposal probabilities denoted as $\widetilde{P}_k(t)$. In the context of Thompson Sampling, we monitor the mean and variance of each arm. The algorithm is formally defined as follows:

Algorithm 2: Thompson Sampling for global maximum

Data: Assuming enough data exist for the each arm so the statistics make sense including μ the mean vector, and σ^2 the variance variance vector, N the counting vector

for $H(t) > \frac{1}{2}H(0)$ **do**
- *Generate a noraml random vector* $V = [v_1, ..v_m]$, where $v_i \sim \mathcal{N}(0, 1)$
- *Find argmax of vector component* $c = \arg\max(\mu + \sigma * V)$
- *Pick a number in the corresponding interval uniformly* $a \in [x_c, x_{c+1}]$
- *Reward* reward $= f(x) + \epsilon(x)$, where $\epsilon(x)$ is a Gaussian noise.
- *Update* μ_c *and* N_c $N_c(t) = N_c(t-1) + 1$, $mu_c = (\text{reward} - \mu_c/N_c$
- *Update* $H(t)$

To update $H(t)$ we need to calculate $\widetilde{P}_k(t)$, which can be obtained by the folling method:

Algorithm 3: Calcualting $\widetilde{P}_k(t)$

For each arm, generate t random samples with mean 0 and variance σ_k^2.

Initializing vector $\widetilde{P}$ **for** $i \leq t$ **do**
- *Find the arm with the highest random sample at time i, denoted as c.*
- $\widetilde{P}_c = \widetilde{P}_c + \frac{1}{t}$

The above method is to approximate $\widetilde{P}_k(t)$ with samples.

When the entropy is defined as:

$$H(t) = \sum_{k=1}^{m} \widetilde{P}_k(t) \log\left(\frac{1}{\widetilde{P}_k(t)}\right)$$

and is small, specifically when $H(t) < \frac{1}{2}H(0)$, we encounter a scenario where further iterations are unlikely to improve results due to the reduced information content. Consequently, repartitioning becomes necessary. To achieve this, we seek the k/m quantiles of the cumulative distribution function (CDF) of the mixture distribution.

An appropriate test function in this context is the simple $x(1-x)$ without noise,

3.3 Result

The following result corresponds to 2 functions with Gaussian noise with mean 0 and variance $(1-x)^2$.

- $f(x) = x(1+x)$
- $f(x) = 2 + 2x(1-x2) + \frac{1}{50}\sin(52\pi x)$

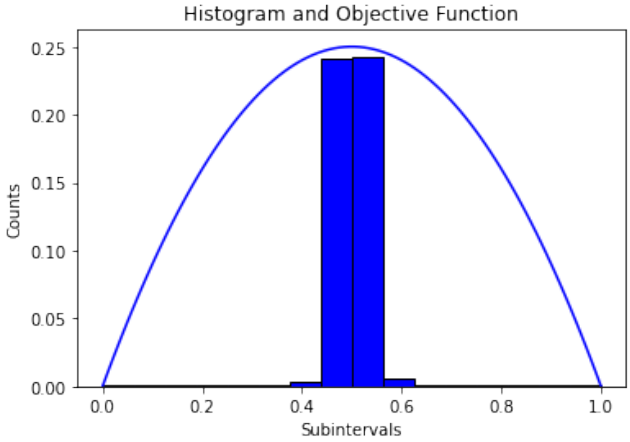

Figure 5: Object function and Histogram with first partition of $f(x) = x(1+x)$

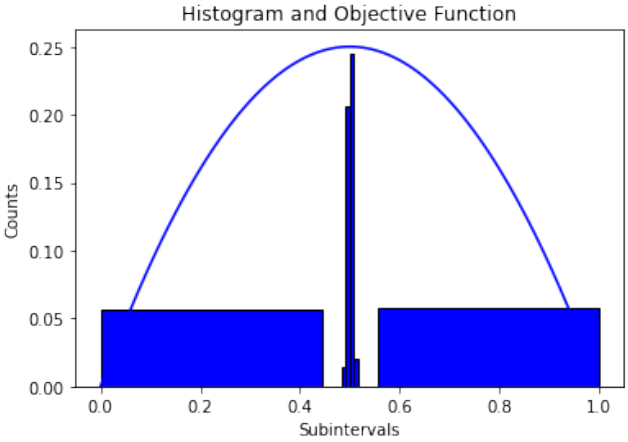

Figure 6: Object function and Histogram with second partition of $f(x) = x(1+x)$

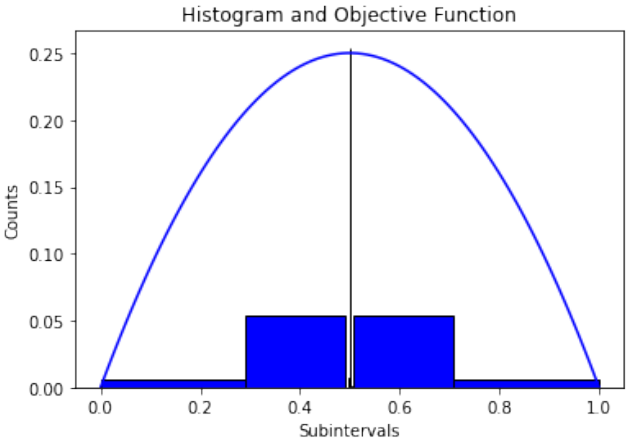

Figure 7: Object function and Histogram with third partition of $f(x) = x(1+x)$

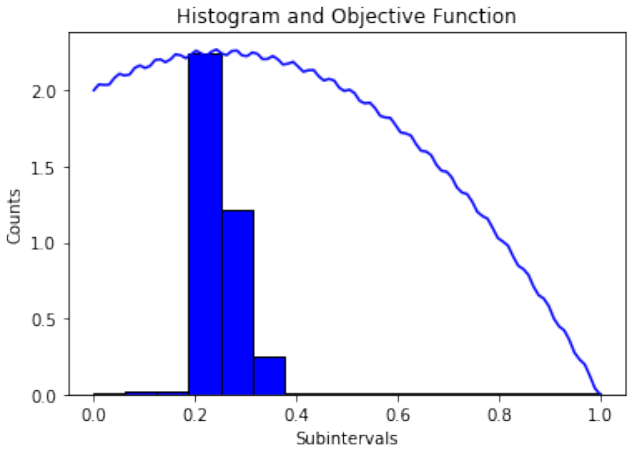

Figure 8: Object function and Histogram with first partition of $f(x) = 2 + 2x(1 - x2) + \frac{1}{50}\sin(52\pi x)$

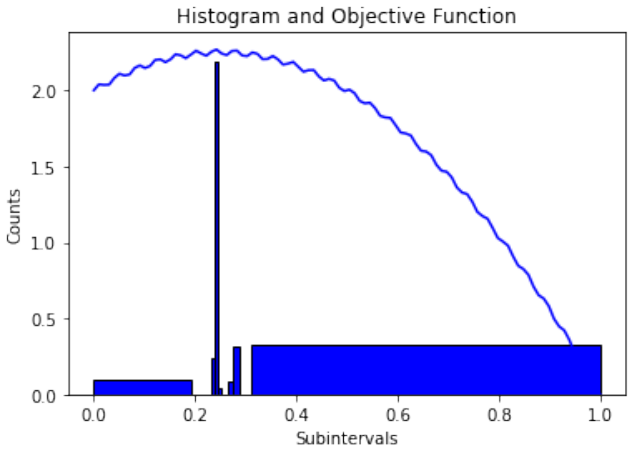

Figure 9: Object function and Histogram with second partition of $f(x) = 2 + 2x(1 - x2) + \frac{1}{50}\sin(52\pi x)$

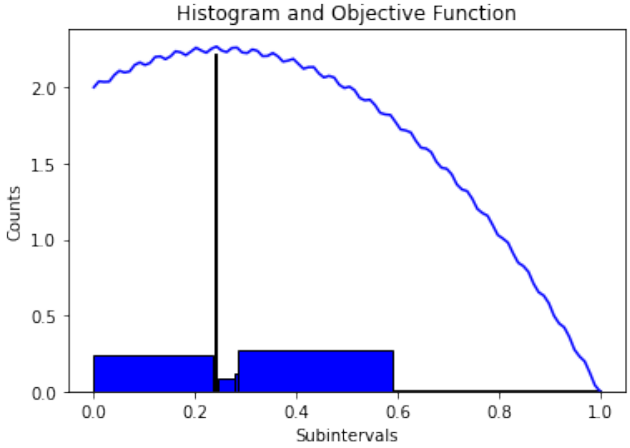

Figure 10: Object function and Histogram with third partition of $f(x) = 2 + 2x(1 - x2) + \frac{1}{50}\sin(52\pi x)$

The subsequent plots illustrate the entropy variation over time:

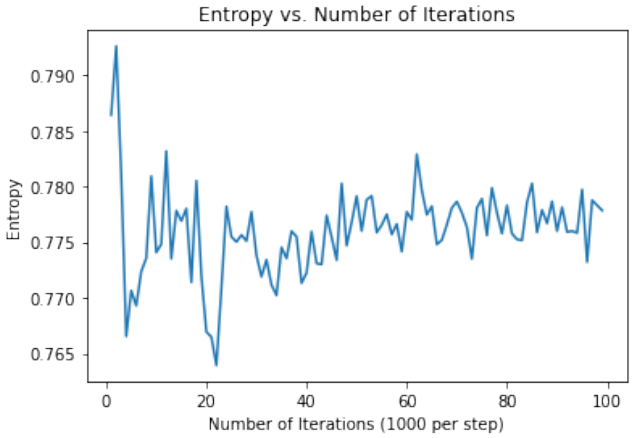

Figure 11: Entropy over time of $f(x) = x(1 - x)$

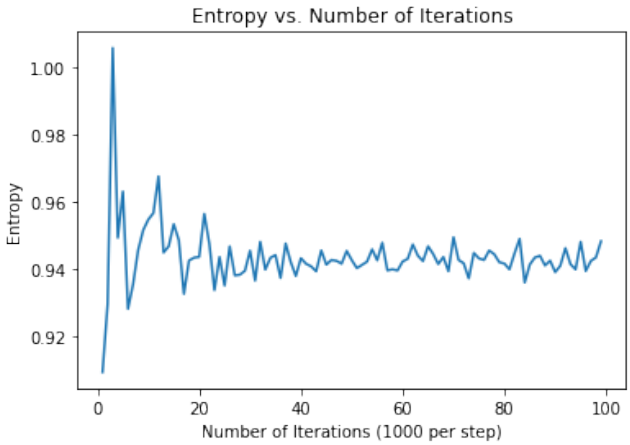

Figure 12: Entropy over time of $f(x) = 2 + 2x(1 - x2) + \frac{1}{50}\sin(52\pi x)$

Note that the entropy experiences decline over time, indicating a convergence. However, it is essential to observe that these diminishing trends are accompanied by oscillations, predominantly caused by the Gaussian noiece. These oscillations may influence the overall stability of the entropy.

4 Conclusion

In conclusion, this project delves into the application of Thompson Sampling in two distinct contexts, aiming to optimize predictive accuracy and maximize payouts in scenarios involving Poisson trials of Bernoulli random variables and to find the global maximum of non-concave functions. Part 1 focuses on the application of Thompson Sampling to coin toss scenarios, demonstrating its efficacy in progressively maximizing cumulative payouts over time. The algorithm's performance is evaluated across various biased coin scenarios, illustrating its adaptability and effectiveness in different conditions.

Moving on to Part 2, the project addresses the challenging task of finding the global maximum of non-concave functions within the unit interval. Utilizing Thompson Sampling with a mixture distribution and partitioning, the algorithm showcases its capability to estimate and approach the global maximum. The results are demonstrated with two test functions, including Gaussian noise, and visualized through histograms and object function plots. The entropy variation over time reveals convergence trends, yet oscillations persist due to inherent noise.

In summary, Thompson Sampling proves to be a versatile and effective algorithm in both predictive scenarios and global maximum search for non-concave functions. Its adaptability and robustness make it a valuable tool in decision-making processes with uncertain outcomes and complex optimization problems.

5 Code

5.1 Part 1

```
n = 100000

def cointoss(p):
    outcomes = [1,0]
    weights = [p,1-p]
    output = random.choices(outcomes,weights)[0]
    return output


def guessing(q1,q0):

    s1 = cointoss(q1)
    s0 = cointoss(q0)


    if s1 > s0:
        return 1
    if s1 == s0:
        return cointoss(0.5)
    else:
        return 0



trail1 = [cointoss(0.6) for i in range(n)]
mw1 = maxwealth(trail1)

trail = trail1
mw = mw1


n1 = 0
n0 = 0

q1 = 0.5
q0 = 0.5

wealth = 0
wealth_history = []


for i in range(len(trail)):
    guess = guessing(q1,q0)
```

```
    reward = int(guess == trail[i])

    wealth += reward

    wealth_history.append(wealth)
    if guess == 1:
        n1 += 1
        q1 = (q1*(n1-1)+reward)/n1
    else:
        n0 += 1
        q0 = (q0*(n0-1)+reward)/n

print(q1,q0,n1,n0,wealth)
```

5.2 Part 2

```
def f1(x):
    return x * (1 - x)

def f2(x):
    return 2+2*x*(1-2*x)+0.02*np.sin(52*math.pi*x)

def eta(x):
    return (1-x)*np.random.normal(0, 1)

def Entropy(P):
    return sum(x*math.log(1/x) for x in P)
```

For iteration = 5000

```
partition = np.arange(0, 1.0625, 1/16)
mu = np.zeros(16)
mu2 = np.zeros(16)
N = np.zeros(16)
iterations = 5000
for i in range(iterations):
    if np.min(N) <= 12:

        a = np.argmin(N)

    else:

        sig = np.sqrt(mu2 - mu**2)
        a = np.argmax(mu + sig * np.random.randn(16))
```

```
    low = partition[a]
    high = partition[a+1]

    x = np.random.uniform(low, high)

    reward = f(x) + 0.0001*eta(x)

    N[a] += 1
    mu[a] += (reward-mu[a])/N[a]
    mu2[a] += (reward**2 - mu2[a]) / N[a]

rand_samples = mu + np.sqrt(mu2 - mu**2) * np.random.randn(iterations, 16)


P = np.zeros(16)


for i in range(iterations):
    am = np.argmax(rand_samples[i])
    P[am] += 1/iterations



P = P + 0.0001 * ((1 / 16) - P)



print("Resulting P vector:", P)
print("Sum of P = " , sum(P))
print("argmax of P = " ,np.argmax(P))
print("N = ", N)
print("Entropy of P = " ,Entropy(P))
```