Computer Science Extended Essay

Topic:

Efficiency and Effectiveness of Algorithms Addressing Exploration-Exploitation Tradeoff

Research Question:

How does the performance of Thompson Sampling compare to that of the Upper Confidence Bound Algorithm in terms of time complexity and regret?

Session: May 2023

Word count: 3724

Table of Contents

1 Introduction

The focus of this essay is to investigate computational complexities and regrets of algorithms addressing the Multi-Armed Bandit Problem (MAPB), a famous problem for analyzing the exploration-exploitation tradeoff. Today, the exploration-exploitation tradeoff dilemma is present in many settings, such as online advertising, healthcare, and finance (A Survey on Practical Applications). This essay will specifically look into Thompson Sampling (TS) and Upper Confidence Bound (UCB) algorithms. These algorithms would be investigated in terms of time complexity—the time taken for an algorithm to run given a set of input values of a certain size—and the total expected regret—the difference between rewards yielded from the optimal action and from the chosen action. Hence, the question: "How does the performance of Thompson Sampling compare to that of the Upper Confidence Bound Algorithm in terms of time complexity and regret?"

1.1 Infrastructure of Exploration-Exploitation Tradeoff Algorithms

The primary goal of exploration-exploitation tradeoff algorithms is to address the dilemma of whether to (exploit) repeat the action that is currently producing the best results or to (explore) make novel decisions in attempt to find a better action (Russo). This type of dilemma occurs most commonly when a learning agent has to make repeated decisions with uncertain rewards.

This exploration-exploitation dilemma holds similarities to real-life contexts. For example, video hosting applications have to decide what type of video to recommend to the user, with the goal of maximizing watch time of the user. In this example, the application would face the dilemma of whether to feed the types of videos that is producing the most watch time or to recommend new types of videos in attempt to find videos that would generate more watch time

(Russo). The application would use solutions similar to the ones presented in this paper to recommend the optimal video.

1.2 Multi-Armed Bandit Problem (MAPB)

The MAPB is a famous problem that allows researchers to analyze the exploration-exploitation tradeoff. In MAPB, there are a finite number of “bandits”, each of which gives a probabilistic reward (LeDoux). Every iteration, the agent chooses one of the bandits and yields its reward. As more iterations are made, the agent gains a better understanding of each bandit’s reward distribution. The objective of the agent is to identify the bandit that yields the highest expected award and exploit it as much as possible (Roberts). This paper explores two different approaches to finding the best bandit and gaining the most cumulative reward in a finite number of actions.

1.3 Gaussian Bandit

There are two main types of bandits: Bernoulli bandits and gaussian bandits. Bernoulli bandits are bandits that outputs binary responses, either a 1 or 0. In the MAPB, there are K bandits, each of which samples from a Bernoulli distribution $D_i, i \in [1, K]$ (Marmerola). However, this paper would be focusing on gaussian bandits, as not many real-world applications involve binary rewards. Gaussian bandits function similar to Bernoulli bandits, except that gaussian bandit samples from a gaussian distribution with expected value μ_i and variance σ_i^2. At each iteration t, the agent selects an action, $a(t)$, and receives a reward of $r(t)$ sampled from $D_{a(t)}$ (Kuleshov). Figure 1 shows an example of the reward distributions of 5 gaussian bandits. In the example, since the red bandit has the highest expected value, the agent should gradually converge towards that bandit.

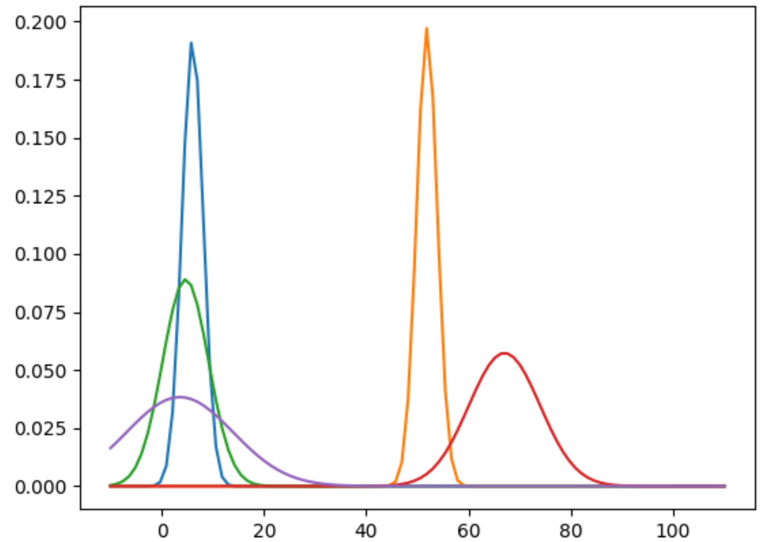

Figure 1: Example distribution of 5 bandits

1.4 Criteria

To compare the performance of two MABP algorithms, one may use several possible criteria to evaluate them. In this paper, 2 criteria would be used:

1. Total expected regret

   At turn T, the total expected regret is defined as

$$R_T = T\mu^* - \sum_{t=1}^{T} \mu_{a(t)}$$

where μ^* is the highest expected reward of all bandits (Kuleshov). In other words, total expected regret is the sum of the differences between the expected reward of the optimal action to the action chosen, so a low regret is considered as having a better performance than a high regret. This metric is used for measuring the ability for the agent to avoid choosing suboptimal bandits. In terms of the example given in section 1.1, a lower regret means recommending videos that yields the most watch time.

2. Run time

Run time denotes the real-world time needed for the agent to decide on an action. This could determine which algorithm is more computationally expensive. In this paper, a low run time is regarded as having better performance than a high run time.

2 Theory

2.1 Thompson Sampling

Thompson sampling is a Bayesian method. The main idea of a Bayesian approach is to combine the prior and likelihood distributions into a posterior distribution that the agent maintains for each bandit. Probability distribution $P(\theta)$, known as prior, could be generated from previous information to model the probability that μ and σ^2 are the actual parameters for each bandit. Likelihood distribution $P(X|\theta)$ model the probability that a new dataset X could be seen given particular values for μ and σ^2. Posterior distribution $P(\theta|X)$ models the probability that particular values for μ and σ^2 are used to generate a given dataset X. The posterior distribution $P(\theta|X) = \frac{P(\theta)P(X|\theta)}{\int P(\theta)P(X|\theta)d\theta}$ can be derived using Bayesian's theorem. In other words, the posterior distribution is simply found by normalizing the product of the prior and likelihood (Kim). To maximize performance, a conjugate prior — a prior where for a given likelihood function (in our case, the likelihood is a gaussian distribution), the posterior is the same distribution as the prior (Conjugate Prior Explained) — should be used. The conjugate prior for a likelihood function of gaussian distribution with unknown mean and variance is gaussian-inverse gamma (Moldovan).

Initially, the agent has no information on the bandits, so it would have to make a guess on the prior. At the start, exploration should be prioritized over exploitation. Therefore, a flat prior (prior distribution as uniform as possible) should be used at the start (Thompson Sampling : Data

Science Concepts). At each iteration, the agent samples from the posterior distribution and selects the action with the highest returned value.

$$a(t) = \underset{a}{\operatorname{argmax}}(P(\theta_a|X_a))$$

Next, the agent can update the posterior from the new data and set it as the prior. Repeating this lets the agent have a more accurate model on each bandit's actual reward distribution. In other words, the posterior would gradually look more similar to the likelihood function.

For each update, the following operations would be made to the parameters of gaussian-inverse gamma.

$$\mu_0 = \frac{\nu\mu_0 + n\bar{x}}{\nu + n}$$

$$\nu = \nu + n$$

$$\alpha = \alpha + \frac{n}{2}$$

$$\beta = \beta + \frac{1}{2}\sum_{i=1}^{n}(x_i - \bar{x})^2 + \frac{n\nu}{\nu + n}\frac{(\bar{x} - \mu_0)^2}{2}$$

where n is the number of observations made after the prior is constructed, and x is the reward yielded at each observation (Conjugate Prior). But since this paper would update the prior immediately after each observation, the operations simply become

$$\mu_0 = \frac{\nu\mu_0 + x}{\nu + 1}$$

$$\nu = \nu + 1$$

$$\alpha = \alpha + \frac{1}{2}$$

$$\beta = \beta + \frac{\nu}{\nu + 1}\frac{(x - \mu_0)^2}{2}$$

where x is the newly yielded reward. α and β are the shape parameter and rate parameter of a gamma distribution. The inverse of a value sampled from the gamma distribution would be used as an estimate of the bandit's variance. This value, along with the estimated expected reward μ_0, would serve as the parameters of a gaussian distribution. Then, the bandit with the highest value sampled from the gaussian gambit would be selected (Roberts).

```
def update(self, r):
        new_mu = (self.mu * self.nu + r)/ (self.nu + 1)
        new_nu = self.nu + 1
        new_alpha = self.alpha + 0.5
        new_beta = self.beta + self.nu * (r - self.mu) ** 2 / (2*(self.nu+1))
        self.mu, self.nu, self.alpha, self.beta = new_mu, new_nu, new_alpha, new_beta
def draw(self):
        if self.nu == 0:
            return float('inf')
        variance = 1/np.random.gamma(self.alpha, 1/self.beta)
        return np.random.normal(self.mu, np.sqrt(variance))
```

The time complexity for an agent to make a decision each iteration is $O(n)$. This is because it needs to sample from all *n* bandits and sampling from a gaussian-inverse gamma function has a time complexity of $O(1)$. Updating the parameters of posterior also has a time complexity of $O(1)$, so in total, Thompson Sampling has a time complexity of $O(n)$.

2.2 Upper Confidence Bound Algorithm (UCB)

In contrast to Thompson Sampling, UCB follows the principle of optimism in the face of uncertainty (Upper Confidence Bound Algorithm in Reinforcement Learning). The principle states that when there is an uncertainty in an action's reward, the agent always assumes the best outcome (Weng). The agent estimates the upper bound of each action using the following equation

$$b_t(a) = Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}$$

where $b_t(a)$ is the upper bound of action a at time t, c is a hyper-parameter that determines how much the agent weighs the uncertainty, Q is the sampling average of the action's reward at time t, and N is the number of times a bandit have been picked at time t. As mentioned, the agent assumes the best outcome, so the agent would choose the action with the highest upper bound. In other words, the chosen action for time t is

$$a(t) = \max_a b_t(a)$$

The equation could be split into 3 terms: exploitation, exploration, and weight of exploration. Each of the 3 terms correspond to $Q_t(a)$, $\sqrt{\frac{\ln t}{N_t(a)}}$, and c in the equation, respectively. In the exploitation term, the higher the rewards yielded from a bandit, the higher the $Q_t(a)$, making the bandit be chosen more often. In the exploration term, $\ln t$ is constantly increasing and $N_t(a)$ is only increased whenever the action is chosen. Therefore, the agent's uncertainty for an action's estimated reward would gradually increase if the action has not been chosen in a while. On the other hand, since $\ln t$ is a log-term and $N_t(a)$ is a linear-term, if both $\ln t$ and $N_t(a)$ increases, the exploitation term decreases. This means that the agent would be more certain of an action's estimated reward after recently choosing it. As shown in Figure 2, when faced with actions with different levels of confidence interval for expected return, the action with the highest upper bound is chosen (The Upper Confidence Bound). In the figure, the center of the box is the exploitation term; the distance between the upper bound and the center is the exploration term multiplied by the weight of exploration.

```
def update(self, r):
        self.sum += r
        self.N += 1
        UCBbandit.t += 1
    def draw(self):
        return self.Q + self.c * np.sqrt(np.log(self.t) / self.N)
    @property
    def Q(self):
        return self.sum / self.N
```

Similar to Thompson Sampling, UCB has a time complexity of $O(n)$ because it needs to calculate the upper confidence bound of each bandit, which has a constant time complexity.

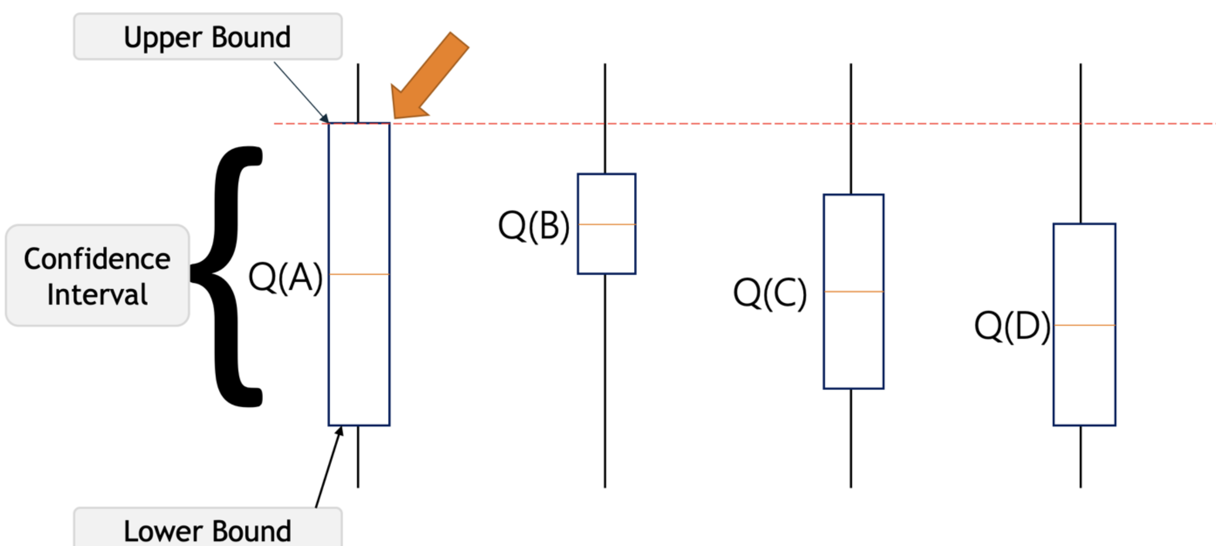

Figure 2: UCB visualized (image from "Upper Confidence Bound Algorithm in Reinforcement Learning." GeeksforGeeks)

2.3 Random Search

Random search is a hyperparameter tuning technique. In machine learning, there are 2 types of parameters: model parameter and hyperparameter. Model parameters are changed during training, which includes the μ, ν, α, and β in Thompson sampling or Q and N in UCB. In contrast, hyperparameters, such as c in UCB, must be tuned before the training for the model to reach optimal performance. Random search utilizes a probability distribution from which the value of the hyperparameter would be sampled. Each iteration, the hyperparameter value is randomly sampled, and the model would be evaluated. The hyperparameter value with the best performance (in terms of regret and run time) would be selected (Jordan).

In this paper, random search sampled from a uniform distribution over the interval (0, 2) for 20 iterations would be used to tune UCB's hyperparameter c. Each hyperparameter's value is evaluated across 5 sets of bandits to ensure that the hyperparameter value does not only perform well on a particular set of bandits.

3 Hypothesis

It is evident that both algorithms have a time complexity of $O(n)$. However, multiple stark contrasts are present in the two algorithms, such as the contrast between Thompson Sampling sampling from probability distributions and UCB performing a mathematical calculation. As a result, while the time complexity might be the same between both algorithms, UCB may have a faster run time than Thompson sampling. This is because sampling from a probability distribution involves multiple mathematical calculations (Boucher).

Since UCB requires the fine-tuning of hyperparameter c, Thompson Sampling may outperform untuned UCB in terms of regret. However, a fine-tuned UCB should outperform Thompson Sampling because the amount that Thompson Sampling explores is dependent on the bandits' variances, resulting in excessive exploration on bandits with large variances. For example, if there are posterior for two bandits as seen in Figure 3 (assume sufficient sampling have been done on both bandits), even though the blue bandit is clearly a better choice, the agent would still choose the yellow bandit approximately 17% of the time due to its large standard variation.

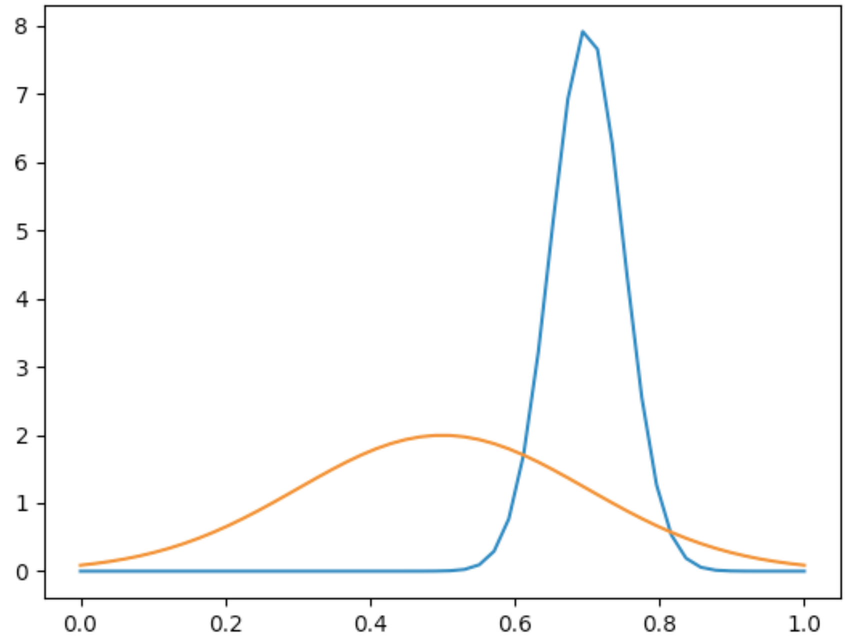

Figure 3: Example of excessive exploration

The experiment will measure the effect of the number of bandits n $(50 \leq n \leq 1000)$ on the run time t and total expected regret R. By varying n, clear relationships between n to t and between n to R should be determined and how these relationships differ between Thompson Sampling and UCB algorithm should be seen.

4 Methodology

4.1 Independent Variable

The independent variable is the number of bandits n. n would be increased in increments of 50, starting from 50 and ending at 1000. This means there would be 20 data points, which is enough data to for a clear relationship to be seen.

4.2 Dependent Variables

This experiment would be measuring two dependent variables: run time and total expected regret of each algorithm. The run time would be measured by taking the difference between the starting and ending *time.perf_counter()*, which is the one of the most precise measure of time in Python. As explained in 1.4, the total expected regret is measured by summing the differences between the expected reward of the optimal action to the action chosen in each iteration.

4.3 Controlled Variables

Variable	**Description**	**Specifications**
Computer and operating system used	The experiment would be conducted on a MacBook Air (13-inch, 2017)	Processor: Intel Core i5@ 1.6 GHz OS: macOS Version 10.14.6 Memory: 8GB RAM
Programming Language Used	The code would be written in Python	Version 3.9.1
Integrated Development Environment	The code would be ran using Jupyter Notebook in Visual Studio Code	
Probability distribution of bandits	All bandits would have a mean uniformly distributed [0, 100] and standard deviation uniformly distributed [0, 25]	RNG: numpy.random.uniform

Table 1: Controlled variables of experiment

4.4 Procedure

1. Set up and import Model.py into main.ipynb
2. Set up a csv.writer for time.csv and regret.csv
3. Set n to 50
4. Create n bandits each with mean uniformly distributed [0, 100] and standard deviation uniformly distributed [0, 25] using the numpy random number generator
5. Instantiate a TS object and two UCB object with the bandits as parameters
6. Run and save the value of UCB.hyperparameter_tuning()
7. Run the change_c() method to tune the hyperparameter of one of the UCB objects
8. Run the *draw* method of each object $10n$ times
9. Record the run time using time.perf_counter() and total expected regret into time.csv and regret.csv, respectively
10. Repeat steps 3-9 increasing n each time by 50 up till n=1000
11. Close time.csv and regret.csv

5 Data Processing and Graph

Below shows the hyperparameter value of untuned and tuned UCB.

UCB Version	Hyperparameter c value
UCB untuned	1
UCB tuned	0.8542355

Table 2: Hyperparameter c values of UCB

Next page shows tables of each algorithm's run time and total expected regret averaged across the 3 trials. Raw data tables could be found in Appendix.

Run time of algorithms			
Number of bandits	Average run time per iteration (seconds)		
	UCB	Thompson Sampling	UCB-tuned
50	0.00026285	0.00029851	0.00025161
100	0.00048617	0.000533	0.00044216
150	0.00069556	0.00077057	0.00065523
200	0.00092323	0.00103402	0.00088882
250	0.00115079	0.00124178	0.00120156
300	0.00175379	0.00162498	0.00147348
350	0.00271818	0.00210755	0.00162437
400	0.00191393	0.00205713	0.0017742
450	0.00218685	0.00230368	0.00200874
500	0.00234424	0.00260733	0.002235
550	0.002578	0.00280712	0.0024294
600	0.00279028	0.00299509	0.00270728
650	0.00304666	0.00331755	0.00291507
700	0.00332581	0.00359158	0.00320431
750	0.00350374	0.00387988	0.00334593
800	0.00414442	0.00419087	0.0036209
850	0.00395936	0.00447484	0.00381179
900	0.00453346	0.00448848	0.00413816
950	0.00428151	0.00460039	0.00410897
1000	0.00455499	0.0050548	0.00456526

Table 3: Table comparing run times

Total expected regret of algorithms			
Number of bandits	Average total expected regret		
	UCB	Thompson Sampling	UCB-tuned
50	2447.14664	2462.50011	2620.72754
100	6386.07381	6948.57884	6583.45591
150	7770.53846	7739.98981	7403.72431
200	10968.6974	11412.1761	12028.6584
250	13195.1884	16646.319	13370.7715
300	17901.786	21101.605	16720.2439
350	21689.4978	22711.9241	20664.263
400	20891.4774	22089.8251	21023.3974
450	27028.1866	29986.9445	28161.7419
500	26379.9599	27819.5647	26401.5187
550	32550.3452	38631.0713	32408.7753
600	32720.9025	35527.8759	31327.4958
650	35540.2956	39982.8684	35502.7526
700	37127.6945	38020.9512	36743.2739
750	40202.9916	45467.4007	41081.4781
800	44944.8528	51311.8922	44515.0518
850	47227.4871	49988.882	45717.6198
900	50588.3703	53290.9091	51891.1016
950	53151.2915	56814.7361	51987.1663
1000	55586.6824	58173.1621	55519.2641

Table 4: Table comparing total expected regret

Plotting these points on a graph and finding a line of best fit for each algorithm gives the following graphs, which compares the scalability of each algorithm for each criterion.

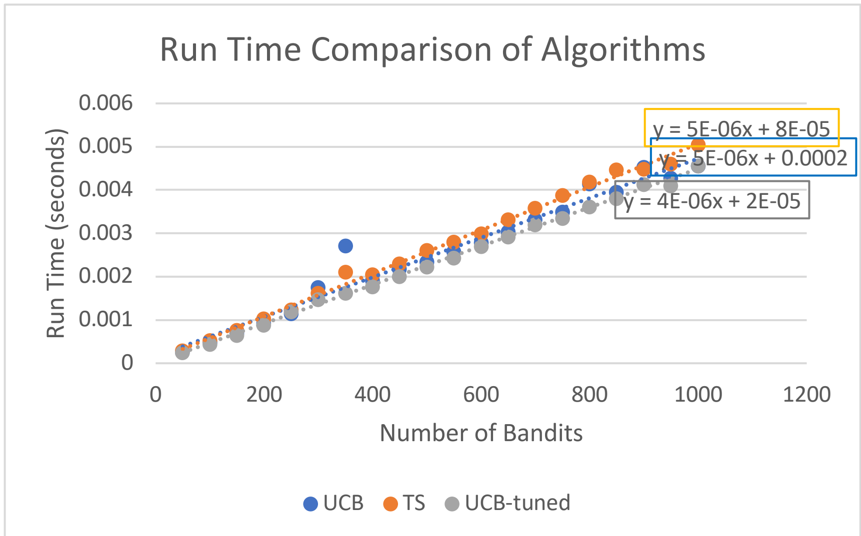

Figure 4: Run time comparison of algorithms

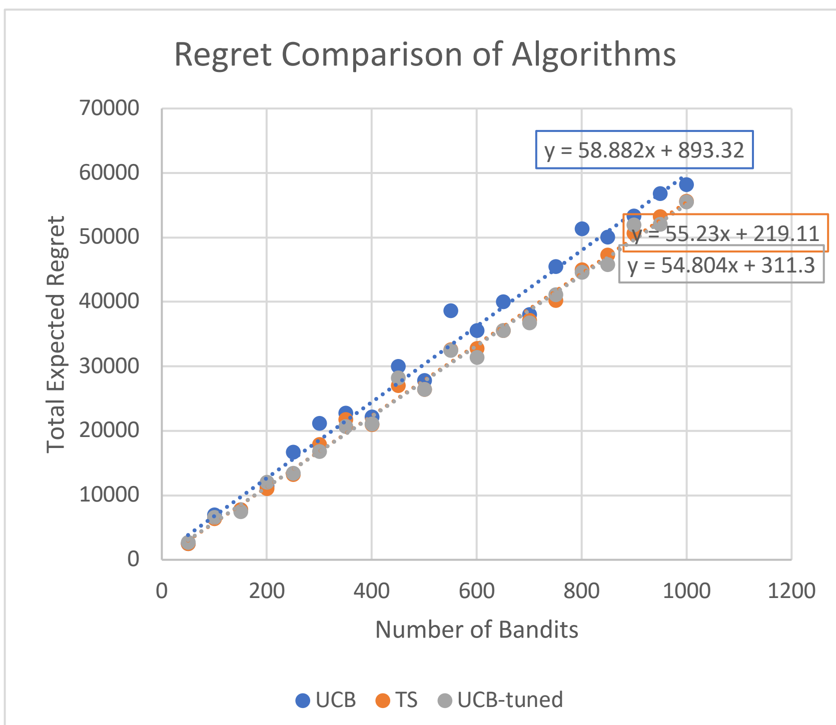

Figure 5: Regret comparison of algorithms

6 Analysis

From the tables and graphs, it is evident that both run time and total expected regret have a linear relationship to the number of bandits. This is also supported by the Pearson correlation coefficient in Table 5, all of which are above 0.95. This suggests that there is a very strong linear relation between the IV and DV's.

Algorithm	Run Time	Total Expected Regret
UCB	0.9616	0.9835
TS	0.9951	0.9948
UCB-tuned	0.9979	0.9928

Table 5: Pearson correlation coefficients for lines of best fit

Since the relations are linear, the slopes of each line of best fit could measure the scalability of each algorithm. Figure 4 compares how the run times scale with the number of bandits. Although the slope value of Thompson sampling and UCB are equal (5×10^{-6}), a more careful look at the graph itself shows that Thompson sampling has a steeper slope. Therefore, in terms of run times, the order of performance of best to worst is: UCB-tuned, UCB, and Thompson sampling. Thompson sampling having a higher run time than both versions of UCB supports my hypothesis that sampling from a probability distribution is more computationally expensive than the UCB equation. However, it is unexpected that UCB (5×10^{-6}) and UCB-tuned (4×10^{-6}) have such drastically different slopes. The only difference in UCB and UCB-tuned is the value of the hyperparameter c, so the calculations should take similar run times. Since UCB was ran before UCB-tuned and the computer used for the experiment was restarted just before the simulation, it was initially thought that the difference is caused by a cold start (program running slow due to startup operations being performed after system reboot). This is not the case, however, as the program was rerun afterwards without restarting the computer and obtained similar results. A possible extension to this paper would be to investigate the cause of this unexpected difference. Moreover, the run-time of UCB-tuned here does not include the time taken to tune the hyperparameter, which took 20.37 seconds to run. If the hyperparameter tuning time was added in, then UCB-tuned would have the slowest run time out of the 3 algorithms.

Similarly, slopes in Figure 5 compares how regret scale with the number of bandits. The slope of the line of best fit for UCB, Thompson sampling, and UCB-tuned are 58.882, 55.23, and 54.804, respectively. Therefore, in terms of total expected regret, the order of performance of best to worst is: UCB-tuned, Thompson sampling, and UCB. This matches the order made in the hypothesis. It is astonishing that even though UCB and UCB-tuned are the same algorithm, simply tuning the hyperparameter can reduce the slope by 7%.

7 Evaluation

The method for this experiment allows for multiple trials for multiple points of the IV, which is the number of bandits. The number of trials conducted ensures that the data collected is reliable and minimizes random error. In addition, two criteria have been used to evaluate the algorithms so that the algorithms can be compared from multiple aspects. The UCB algorithm, which consists of a hyperparameter, has been tested two versions: one with untuned c and one with tuned c; this ensures that a fair comparison could be made between Thompson sampling. Table 5 shows that the Pearson's correlation coefficients are all above 0.95, meaning that a line of best fit could be used to extrapolate the data and measure an algorithm's scalability.

7.1 Limitations

However, random and systematic errors are present in the data. As seen from the graphs, there are a few points that show significant deviation from the line of best fit, such as (350, 0.0027) in Figure 4 and (550, 38631) in Figure 5. The outlier in run time may be caused by varying processing consumption affecting the CPU's clock speed. The outlier in regret could be caused by the exploration term in UCB. Since the upper bound for a bandit have not been recently picked would gradually increase, very suboptimal bandits (a bandit that yields low rewards) would soon overtake the upper bound of more optimal bandits, which causes an abnormal spike in regret. Moreover, the y-intercepts of the lines of best fit are not zero. Theoretically, if there are no bandits, the agents should take 0 run time to make a decision and should have 0 regret. The error in run time could be caused by the Python interpreter having to go through each line of code and converting them into code that the machine could understand, and the error in total expected regret could be caused by the random nature of the multi-armed bandit problem.

As mentioned in section 3, Thompson sampling may be prone to excessive exploration. A potential improvement would be to treat the likelihood function like a gaussian distribution with unknown mean and known variance. By doing so, the conjugate prior would become a gaussian distribution with parameter μ_0, σ_0 (Conjugate Prior). Here, the update for mean μ_0 is similar to the update equation in section 2.1. The main difference lies in the variance σ_0. Instead of using an inverse gamma distribution to estimate the variance of the bandit, the variance would decrease the more the bandit is chosen. This can potentially decrease the exploration on suboptimal bandits.

8 Conclusion

This experiment implemented the theory behind Upper Confidence Bound (UCB) and Thompson Sampling as explained in section 2 and applied them to the Multi-Armed Bandit Problem to compare their performances in terms of run time and total expected regret. Plotting number of bandits vs run time and number of bandits vs total expected regret gave me lines of best fits that could compare the scalability of each algorithm. It is concluded that in terms of run time, UCB always performs better than Thompson sampling, but it is surprising that UCB-tuned was noticeably faster than UCB, as seen from their respective slopes in Figure 4. As expected, there is a linear relationship between number of bandits and run time, which is also apparent in Figure 4. It should be noted, however, that the run time here does not take into account for the hyperparameter tuning time. In terms of total expected regret, as seen in Figure 5, UCB only performs better than Thompson sampling when the hyperparameter c is tuned, which could be done using hyperparameter tuning algorithms such as random search. For large enough number of bandits, it is possible that the run time difference between UCB and Thompson sampling would be great enough so that tuning the hyperparameter would still result in UCB still having a faster total run, which makes UCB the better algorithm in both criteria for large number of bandits.

9 Further Scope

This experiment was only done on gaussian bandits. On the other hand, real-world applications may not be limited to gaussian distribution; for example, an Ad feeder usually has Bernoulli bandits. Since the performance of algorithms may vary on different likelihood distributions, the results of this experiment may not be extrapolated to other likelihoods. Therefore, if given more time, a possible extension to this experiment would be to investigate the performance of UCB and Thompson Sampling on other likelihood functions.

In addition, the number of IV points and trials conducted was limited by the computational power of the computer and time. If a more powerful computer was used and more time was given, more trials could be conducted on a greater range of bandit count. Although the results yielded from more trials may not vary much from the results of this experiment, it would make the data more reliable. Also, a more powerful computer could allow the testing of many other exploration-exploitation algorithms, such as decayed epsilon greedy and Boltzmann exploration.

Works Cited

Boucher, Christopher. "Sampling Random Numbers from Probability Distribution Functions." *COMSOL*, www.comsol.com/blogs/sampling-random-numbers-from-probability-distribution-functions/.

Jordan, Jeremy. "Hyperparameter Tuning for Machine Learning Models." *Jeremy Jordan*, 2 Nov. 2017, www.jeremyjordan.me/hyperparameter-tuning/.

Kim, Aerin. "Bayesian Inference ? Intuition and Example." *Towards Data Science*, 2 Jan. 2020, towardsdatascience.com/bayesian-inference-intuition-and-example-148fd8fb95d6.

---. "Conjugate Prior Explained." *Towards Data Science*, 8 Jan. 2020, towardsdatascience.com/conjugate-prior-explained-75957dc80bfb.

Kuleshov, Volodymyr, and Doina Precup. *Algorithms for the Multi-armed Bandit Problem*. Edited by Leslie Pack Kaelbling, Arxiv. *Arxiv*, arxiv.org/pdf/1402.6028.pdf.

LeDoux, James. "Multi-Armed Bandits in Python: Epsilon Greedy, UCB1, Bayesian UCB, and EXP3." *jamesrledoux*, 24 Mar. 2020, jamesrledoux.com/algorithms/bandit-algorithms-epsilon-ucb-exp-python/.

Marmerola, Guilherme Duarte. "Introduction to Thompson Sampling: The Bernoulli Bandit." *GitHub*, 21 Nov. 2017, gdmarmerola.github.io/ts-for-bernoulli-bandit/.

Moldovan, Teodor Mihai. "The Conjugate Prior for the Normal Distribution." *Berkeley*, 8 Feb. 2010, people.eecs.berkeley.edu/~jordan/courses/260-spring10/lectures/lecture5.pdf.

Roberts, Steve. "Thompson Sampling." *Towards Data Science*, 2 Nov. 2020, towardsdatascience.com/thompson-sampling-fc28817eacb8.

---. "Thompson Sampling Using Conjugate Priors." *Towards Data Science*, 9 Mar. 2021, towardsdatascience.com/thompson-sampling-using-conjugate-priors-e0a18348ea2d.

---. "The Upper Confidence Bound (UCB) Bandit Algorithm." *Towards Data Science*, 26 Oct. 2020, towardsdatascience.com/the-upper-confidence-bound-ucb-bandit-algorithm-c05c2bf4c13f.

Russo, Daniel J., et al. *A Tutorial on Thompson Sampling*. Hanover, 2018. *Arxiv*, arxiv.org/pdf/1707.02038.pdf?ref=hackernoon.com.

A Survey on Practical Applications of Multi-Armed and Contextual Bandits. 2 Apr. 2019, arxiv.org/abs/1904.10040.

"Thompson Sampling : Data Science Concepts." *Youtube*, 7 July 2021, www.youtube.com/watch?v=Zgwfw3bzSmQ&feature=youtu.be. Accessed 15 Dec. 2021.

"Upper Confidence Bound Algorithm in Reinforcement Learning." *GeeksforGeeks*, 19 Feb. 2020, www.geeksforgeeks.org/upper-confidence-bound-algorithm-in-reinforcement-learning/.

Weng, Lilian. "The Multi-Armed Bandit Problem and Its Solutions." *GitHub*, 23 Jan. 2018, lilianweng.github.io/lil-log/2018/01/23/the-multi-armed-bandit-problem-and-its-solutions.html#bandit-strategies.

10 Appendix

Appendix A Full regret data

Total expected regret of algorithms									
# of bandits	UCB			Thompson Sampling			UCB-tuned		
	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3
50	0.0002712	0.00034207	0.00028224	0.00025822	0.00025575	0.00027457	0.00024605	0.00026093	0.00024785
100	0.00058074	0.00050211	0.00051616	0.00050398	0.00047047	0.00048407	0.00044391	0.00045152	0.00043105
150	0.00077074	0.00078465	0.00075631	0.00068951	0.00069955	0.00069761	0.00065802	0.00065342	0.00065427
200	0.00102953	0.0009811	0.00109141	0.00095629	0.00091467	0.00089873	0.00089965	0.00087425	0.00089257
250	0.00124726	0.00123797	0.00124011	0.00116488	0.0011233	0.00116418	0.00112817	0.00111491	0.0013616
300	0.00164214	0.00150315	0.00172965	0.00173002	0.00139033	0.00214103	0.00132454	0.00133596	0.00175995
350	0.00263006	0.00184419	0.0018484	0.00486038	0.00165672	0.00163746	0.00161104	0.00159605	0.00166602
400	0.00203609	0.00203904	0.00209625	0.00200124	0.00186008	0.00188048	0.0017786	0.00177586	0.00176813
450	0.00235485	0.00229349	0.0022627	0.00227586	0.00217994	0.00210474	0.00198427	0.00206401	0.00197793
500	0.00262383	0.0025278	0.00267036	0.00232244	0.00234269	0.0023676	0.00223628	0.00220032	0.00226839
550	0.00279723	0.00277943	0.00284471	0.00262691	0.00258682	0.00252026	0.00242922	0.00245393	0.00240505
600	0.00305943	0.00294024	0.00298561	0.00280139	0.0027876	0.00278186	0.00273219	0.00263066	0.00275899
650	0.00330242	0.00336723	0.003283	0.00305572	0.00301652	0.00306773	0.00290065	0.00293671	0.00290784
700	0.00362108	0.00355851	0.00359516	0.00344179	0.00324232	0.00329332	0.0033917	0.00308444	0.00313679
750	0.00391861	0.00380231	0.0039187	0.00349112	0.00347855	0.00354156	0.00336105	0.0033238	0.00335293
800	0.00411795	0.00410647	0.00434819	0.00371565	0.00376196	0.00495565	0.00362556	0.00363731	0.00359983
850	0.0047762	0.00414203	0.00450628	0.00405117	0.00385683	0.00397008	0.00366105	0.00369657	0.00407775
900	0.00455385	0.00417031	0.00474129	0.004754	0.00465387	0.00419252	0.00462981	0.00388963	0.00389503
950	0.00433135	0.00473584	0.00473397	0.00438296	0.00416184	0.00429974	0.0040956	0.00410427	0.00412705
1000	0.00517161	0.00493706	0.00505574	0.00452192	0.00463034	0.00451272	0.00430595	0.00485239	0.00453745

Appendix B Full run time data

Run time of algorithms									
# of bandits	UCB			Thompson Sampling			UCB-tuned		
	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3	Trial 1	Trial 2	Trial 3
50	0.00025822	0.00025575	0.00027457	0.0002712	0.00034207	0.00028224	0.00024605	0.00026093	0.00024785
100	0.00050398	0.00047047	0.00048407	0.00058074	0.00050211	0.00051616	0.00044391	0.00045152	0.00043105
150	0.00068951	0.00069955	0.00069761	0.00077074	0.00078465	0.00075631	0.00065802	0.00065342	0.00065427
200	0.00095629	0.00091467	0.00089873	0.00102953	0.0009811	0.00109141	0.00089965	0.00087425	0.00089257
250	0.00116488	0.0011233	0.00116418	0.00124726	0.00123797	0.00124011	0.00112817	0.00111491	0.0013616
300	0.00173002	0.00139033	0.00214103	0.00164214	0.00150315	0.00172965	0.00132454	0.00133596	0.00175995
350	0.00486038	0.00165672	0.00163746	0.00263006	0.00184419	0.0018484	0.00161104	0.00159605	0.00166602
400	0.00200124	0.00186008	0.00188048	0.00203609	0.00203904	0.00209625	0.0017786	0.00177586	0.00176813
450	0.00227586	0.00217994	0.00210474	0.00235485	0.00229349	0.0022627	0.00198427	0.00206401	0.00197793
500	0.00232244	0.00234269	0.0023676	0.00262383	0.0025278	0.00267036	0.00223628	0.00220032	0.00226839
550	0.00262691	0.00258682	0.00252026	0.00279723	0.00277943	0.00284471	0.00242922	0.00245393	0.00240505
600	0.00280139	0.0027876	0.00278186	0.00305943	0.00294024	0.00298561	0.00273219	0.00263066	0.00275899
650	0.00305572	0.00301652	0.00306773	0.00330242	0.00336723	0.003283	0.00290065	0.00293671	0.00290784
700	0.00344179	0.00324232	0.00329332	0.00362108	0.00355851	0.00359516	0.0033917	0.00308444	0.00313679
750	0.00349112	0.00347855	0.00354156	0.00391861	0.00380231	0.0039187	0.00336105	0.0033238	0.00335293
800	0.00371565	0.00376196	0.00495565	0.00411795	0.00410647	0.00434819	0.00362556	0.00363731	0.00359983
850	0.00405117	0.00385683	0.00397008	0.0047762	0.00414203	0.00450628	0.00366105	0.00369657	0.00407775
900	0.004754	0.00465387	0.00419252	0.00455385	0.00417031	0.00474129	0.00462981	0.00388963	0.00389503
950	0.00438296	0.00416184	0.00429974	0.00433135	0.00473584	0.00473397	0.0040956	0.00410427	0.00412705
1000	0.00452192	0.00463034	0.00451272	0.00517161	0.00493706	0.00505574	0.00430595	0.00485239	0.00453745

Appendix C Main.ipynb

Cell 1

```
from Model import np, TS, UCB, Bandit, TS2
from matplotlib import pyplot as plt
from tqdm.notebook import tqdm
import csv
from time import perf_counter

np.random.seed(0)
```

Cell 2

```
best_c = UCB.hyperparameter_tuning(20)
print("Best c for UCB:", best_c)
```

Cell 3

```
def run(alg, n, T=None, *, bandits=None, loading=False, name=None):
    bandits = [Bandit(i) for i in range(n)] if bandits is None else bandits
    optimal = max(bandits, key=lambda index: index.mean).mean
    if T is None:
        T = n * 10
    regret = 0
    alg = alg(n, bandits)
    if name == "UCB tuned":
        alg.change_c(best_c)
    regrets = [0]
    bandit_count = {i:0 for i in bandits}
    time = 0
    for i in tqdm(range(T)) if loading else range(T):
        start = perf_counter()
        bandit, reward = alg.draw()
        end = perf_counter()
        time += end-start
        bandit_count[bandit] += 1

        regret += optimal - bandit.mean
        regrets.append(regret)
    time /= T
    plt.plot(range(T+1), regrets, label=alg.name if name is None else name)
    return bandits, time, regret
```

Cell 4

```
for n in range(50, 1001, 50):
    bandits = [Bandit(i) for i in range(n)]
    optimal = max(bandits, key=lambda index: index.mean)
    print(optimal)
```

```
    rUCB, rTS, rUCBt = [], [], []
    tUCB, tTS, tUCBt = [], [], []
    try:
        for _ in range(3):
            _, timeUCB, regretUCB = run(UCB, n, bandits=bandits)
            _, timeTS, regretTS = run(TS2, n, bandits=bandits)
            _, timeUCBt, regretUCBt = run(UCB, n, bandits=bandits, name="UCB tuned")
            rUCB.append(regretUCB)
            rTS.append(regretTS)
            rUCBt.append(regretUCBt)
            tUCB.append(timeUCB)
            tTS.append(timeTS)
            tUCBt.append(timeUCBt)
    except KeyboardInterrupt:
        time_writer.writerow([n] +tUCB + tTS + tUCBt + [np.average(tUCB),
np.average(tTS), np.average(tUCBt)])
        regret_writer.writerow([n] +tUCB + tTS + tUCBt+[np.average(rUCB),
np.average(rTS), np.average(rUCBt)])

    time_writer.writerow([n] +tUCB + tTS + tUCBt + [np.average(tUCB), np.average(tTS),
np.average(tUCBt)])
    regret_writer.writerow([n] +tUCB + tTS + tUCBt+[np.average(rUCB), np.average(rTS),
np.average(rUCBt)])
file1.close()
file2.close()
```

Appendix D Model.py

```
import numpy as np
from tqdm import tqdm

class Algorithm:
    def __init__(self, n, bandits=None, *, alg) -> None:
        self.n = n
        self.bandits = [Bandit() for _ in range(n)] if bandits is None else bandits
        self.params = [alg(bandit) for bandit in self.bandits]
    def draw(self, update=True):
        bandit_n = max(range(self.n), key=lambda index: self.params[index].draw())
        bandit = self.bandits[bandit_n]
        reward = bandit.draw()
        if update:
            self.params[bandit_n].update(reward)
        return bandit, reward

class TS(Algorithm):
    name = "TS"
    def __init__(self, n, bandits=None) -> None:
```

```python
        super().__init__(n, bandits, alg=TSbandit)


class UCB(Algorithm):
    name = "UCB"
    def __init__(self, n, bandits=None) -> None:
        super().__init__(n, bandits, alg=UCBbandit)
        UCBbandit.t = 1
    def change_c(self, c):
        for i in self.params:
            i: UCBbandit
            i.c = c

    @classmethod
    def hyperparameter_tuning(self, n, num_sets=5):
        bandits = [[Bandit(i) for i in range(np.random.randint(50, 1000))] for _ in
range(num_sets)]
        points = np.random.uniform(0, 2, n)
        max_reward, max_ucb = -float('inf'), None
        for _ in tqdm(range(n)):
            reward = 0
            for i in range(num_sets):
                ucb = UCB(len(bandits[i]), bandits[i])
                ucb.change_c(points[i])
                for _ in range(ucb.n):
                    _, r = ucb.draw()
                    reward += r
            if reward > max_reward:
                max_reward = reward
                max_ucb = points[i]
        return max_ucb


class UCBbandit:
    t = 1
    def __init__(self, bandit) -> None:
        self.bandit = bandit
        self.sum = 0
        self.N = 0.0001
        self.c = 1
    def update(self, r):
        self.sum += r
        self.N += 1
        UCBbandit.t += 1
    def draw(self):
        return self.Q + self.c * np.sqrt(np.log(self.t) / self.N)
    @property
    def Q(self):
```

```python
        return self.sum / self.N


class TSbandit:
    def __init__(self, bandit) -> None:
        self.bandit = bandit
        self.mu = 0
        self.alpha = 0.5
        self.beta = 1
        self.nu = 0
    def update(self, r):
        new_mu = (self.mu * self.nu + r)/ (self.nu + 1)
        new_nu = self.nu + 1
        new_alpha = self.alpha + 0.5
        new_beta = self.beta + self.nu * (r - self.mu) ** 2 / (2*(self.nu+1))
        self.mu, self.nu, self.alpha, self.beta = new_mu, new_nu, new_alpha, new_beta
    def draw(self):
        if self.nu == 0:
            return float('inf')
        variance = 1/np.random.gamma(self.alpha, 1/self.beta)
        return np.random.normal(self.mu, np.sqrt(variance))

class Bandit:
    N = 0
    def __init__(self, n, mean=None, std=None) -> None:
        self.mean = np.random.uniform(0, 100) if mean is None else mean
        self.std = np.random.uniform(0, 25) if std is None else std
        self.n = n
        self.N = n
    def draw(self):
        return np.random.normal(self.mean, self.std)
    def __str__(self) -> str:
        return f"Bandit {self.n}: mean {self.mean} std {self.std}"
```