Bandit Algorithms

Brandon Yang
University of Virginia
Jqm9ba@virginia.edu

***Abstract*—This is a report on different bandit algorithms and their implementations. This report consists of both Multi-Armed Bandits and Contextual Linear Bandits.**

I. Multi-armed Bandit Problem

A. Epsilon-Greedy

Epsilon-Greedy is a random exploration strategy. At each timestep, with probability ϵ the algorithm chooses to pick a random arm to play, and with probability $1-\epsilon$, the algorithm chooses to exploit the current best arm, which is calculated as the arm with the highest mean value μ_i, defined as follows:

$$\mu_i = \frac{\mu_i n_i + r_i}{n_i + 1}$$

n_i is the number of times the algorithm picked arm i, r_i is the reward of arm i. Another strategy in Epsilon-Greedy is to define epsilon as a decay function, known as epsilon decay. This way, the algorithm would have higher values of epsilon in the beginning, and lower values of epsilon as $t \to T$. The following decay function is used:

$$explore = \mathcal{B}(1, (\mathrm{n}+1)^{\frac{-1}{3}})$$

When explore is 1, the algorithm would choose a random arm, otherwise, it will exploit the best current arm. A code snippet is provided in Appendix A.

Algorithm 1 Epsilon-Greedy Multi-Armed Bandit

0: Inputs: d = num_arms, $\epsilon = 0.1$
1: $\mu = 0_d, n = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: **with** $P = \epsilon$:
4: Play random arm
5: **with** $P = 1 - \epsilon$:
6: Play arm $a := \arg\max \mu_i$ and observe reward r_i
7: Update $\mu_i = \frac{\mu_i n_i + r_i}{n_i + 1}$, $n_i = n_i + 1$
8: **end for**

B. Upper Confidence Bound (UCB)

UCB is often phrased as optimism in the face of uncertainty. At each timestep, the algorithm receives a reward function r_i when arm i is pulled, which in turn creates a reward distribution. To calculate the upper bound, the confidence interval given round of n trials is computed as follows:

$$confidence\ interval = \alpha \sqrt{\frac{2\ln(n)}{n_i}}$$

α is a tunable hyperparameter that puts a weight onto the confidence interval, it is initialized to 0.25 in this project. n is the number of timesteps, n_i is the number of times the algorithm picked arm i. The upper confidence bound is simply the sum of μ_i and the confidence interval, meaning at each time step, we want to pick the arm that maximizes UCB_{score}, computed as follows:

$$UCB_{score} = \mu_i + \alpha \sqrt{\frac{2\ln(n)}{n_i}}$$

Algorithm 2 Upper Confidence Bound Multi-Armed Bandit

0: Inputs: d = num_arms, $\alpha = 0.25$
1: $\mu = 0_d, n = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: Play all arms once
4: Play arm $a := \arg\max \mu_i + \alpha \sqrt{\frac{2\ln(n)}{n_i}}$
5: Update $\mu_i = \frac{\mu_i n_i + r_i}{n_i + 1}$, $n_i = n_i + 1$
6: **end for**

C. Thompson Sampling (TS)

Thompson Sampling uses a Bayesian statistical approach to model the reward distribution of arms. In this paper, we assume reward is Gaussian with unknown mean and known standard deviation, which means that this implementation of Thompson Sampling consists of a Gaussian posterior. Frist, all the arms are played once to generate their respective mean μ_i and standard deviation σ_i. The mean and standard deviation is formulated as follows:

$$\mu_i = \frac{\mu_i n_i + r_i}{n_i + 1}$$

$$\sigma_i = \frac{1}{n_i}$$

n_i is the number of times arm i was picked. After all the arms are pulled once, the algorithm would normally sample from the posterior and pick the arm that generates the greatest value from their respective distributions in the posterior:

$$\arg\max \mathcal{N}(\mu_i, \sigma_i)$$

Algorithm 3 Thompson Sampling Multi-Armed Bandit

0: Inputs: d = num_arms
1: $\mu = 0_d, \sigma = 0_d, n = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: Play all arms once
4: Play arm $a := \arg\max \mathcal{N}(\mu_i, \sigma_i)$
5: Update $\mu_i = \frac{\mu_i n_i + r_i}{n_i + 1}$, $\sigma_i = \frac{1}{n_i}$, $n_i = n_i + 1$
6: **end for**

D. **Perturbed-History Exploration (PHE)**

PHE approaches the problem by implementing pseudo-rewards in addition to the actual rewards. At each timestep, PHE pulls the arm with highest average reward in perturbed history, and computes its pseudo reward as follows:

$$pseudo\_rewards = (Z_l)_{l=1}^{as} \sim Ber(0.5)$$

a is a tuneable integer $a > 0$, and s is the number of pulls arm i has in the first t rounds: $s = T_{i,t-1}$. For example, if a is 0.5, and the number of times arm i has been visited is $s = 11$, then we want sample from the Bernoulli distribution 6 times, because we will take the ceiling value of the multiple between a and s: $Ceil(11 \times 0.5) = 6$. The sum of the values generated from the distribution is the pseudo reward $U_{i,s}$. At each timestep, the pseudo reward is added in addition to the actual reward history of the arm, $V_{i,s} + U_{i,s}$, and the average value of each arm is calculated as follows:

$$\mu_i = \frac{V_{i,s} + U_{i,s}}{(a+1)s}$$

The algorithm will compute all values of μ_i for each arm i, and chooses to play the arm that outputs the greatest value of μ_i, and receives a reward r_i as a result, which will be added to its sum of cumulative reward after s pulls: $V_{i,s} = V_{i,t-1} + r_i$. The process is repeated with updated value of $T_{i,t}$, which is the number of times arm i has been visited.

Algorithm 4 PHE Multi-Armed Bandit

0: Inputs: d = num_arms, $a = 0.5$
1: Initialize: $\mu = 0_d$, $s = 0_d$, $T = 0_d$, $V = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: **if** $T_{i,t-1} > 0$ **then**
4: $s = T_{i,t-1}$
5: $(Z_l)_{l=1}^{as} \sim Ber(0.5)$
6: $U_{i,s} = \sum_{l=1}^{as} Z_l$
7: $\mu_i = \frac{V_{i,s}+U_{i,s}}{(a+1)s}$
8: **else**
9: $\mu_i = \infty$
10: Play arm $:=$ arg max μ_i and observe reward r_i
11: Update $T_{i,t} = T_{i,t-1} + 1$, $V_{i,t} = V_{i,t-1} + r_i$
12: **end for**

E. **Multi-Armed Bandit Algorithms Analysis**

The analysis portion for multi-armed problems consists of four algorithms: Epsilon Greedy, UCB, Thompson Sampling, and PHE. The following algorithm hyperparameters were used for all experiment runs:

Epsilon-Greedy: $\epsilon = (\mathrm{n}+1)^{\frac{-1}{3}}$, n = timestep
UCB: $\alpha = 0.25$
PHE: $\alpha = 0.5$

The following experiment hyperparameters remain unchanged for all experiment runs:

Testing iterations: 15,000 steps

Number of users: 10

The experiment consists of different values of Gaussian noise scale and number of arms (K), results are shown below:

Constant Gaussian noise scale: 0.1, K = 15, 25, 50, 100, 300

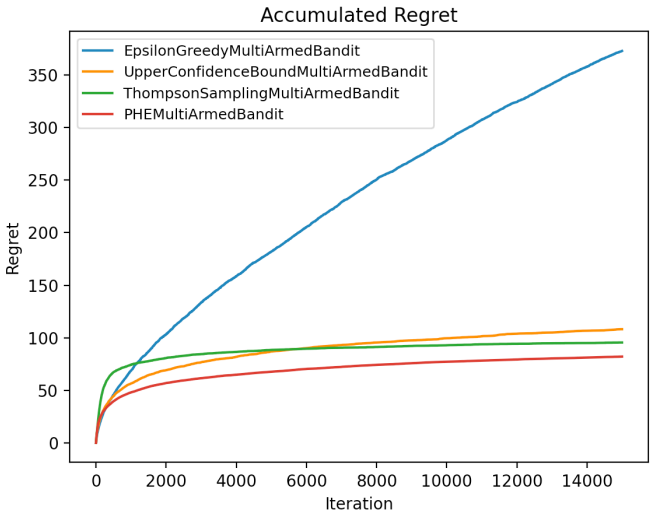


Figure 1: noise = 0.1, K = 15

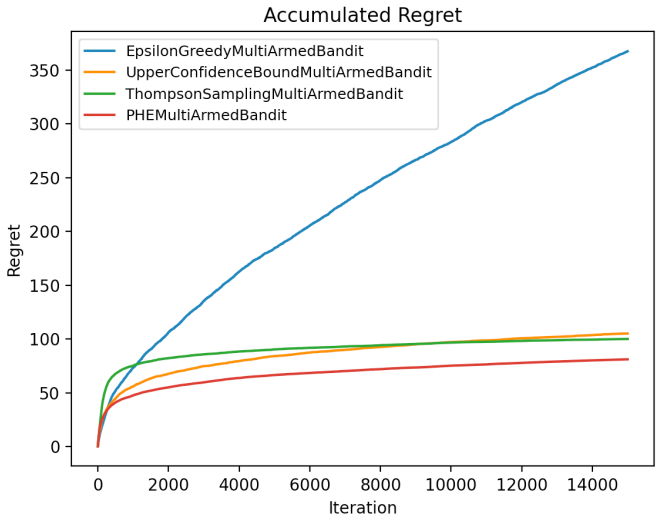


Figure 2: noise = 0.1, K = 25

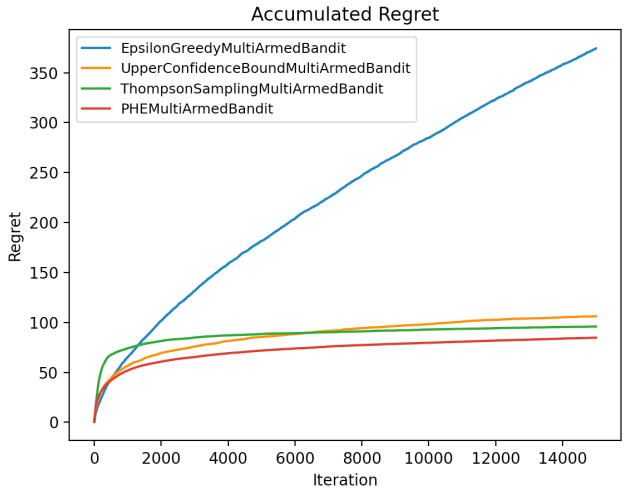


Figure 3: noise = 0.1, K = 50

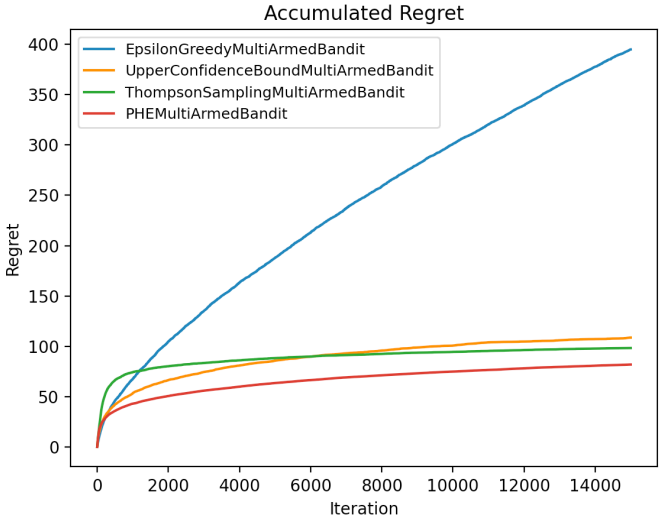


Figure 4: noise = 0.1 K = 100

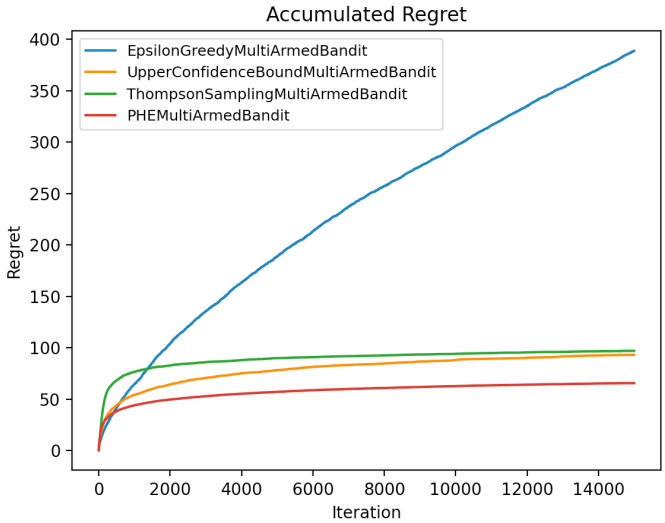


Figure 5: noise = 0.1, K = 300

Constant K = 25, Gaussian noise scale = 0.03, 0.05, 0.25, 0.50, 1.0

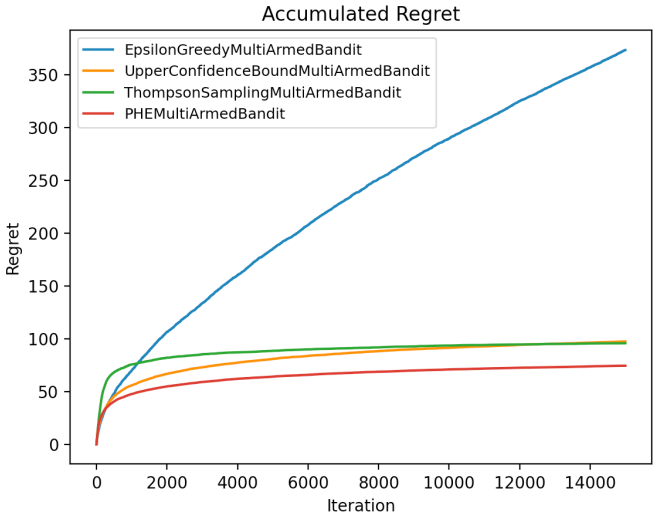


Figure 6: noise = 0.03, K = 25

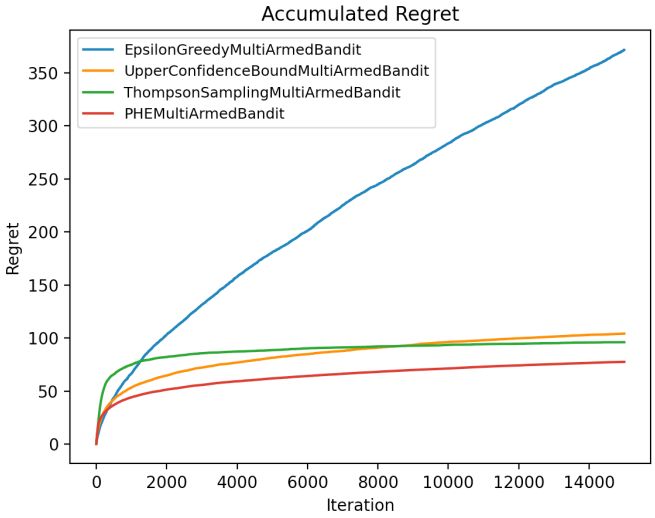


Figure 7: noise = 0.05, K = 25

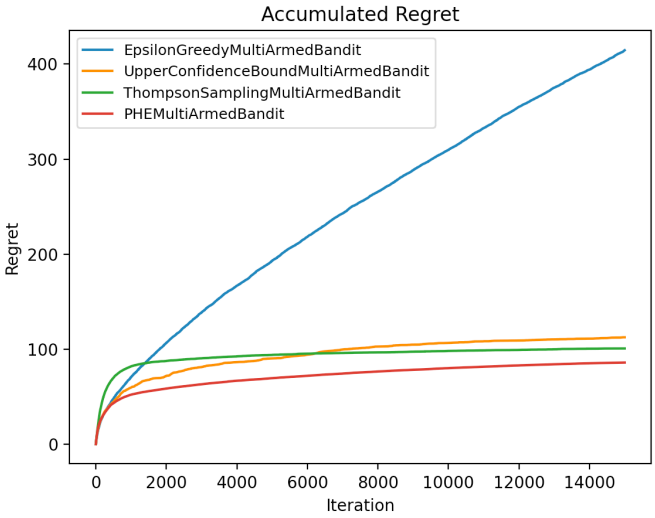


Figure 8: noise = 0.25, K = 25

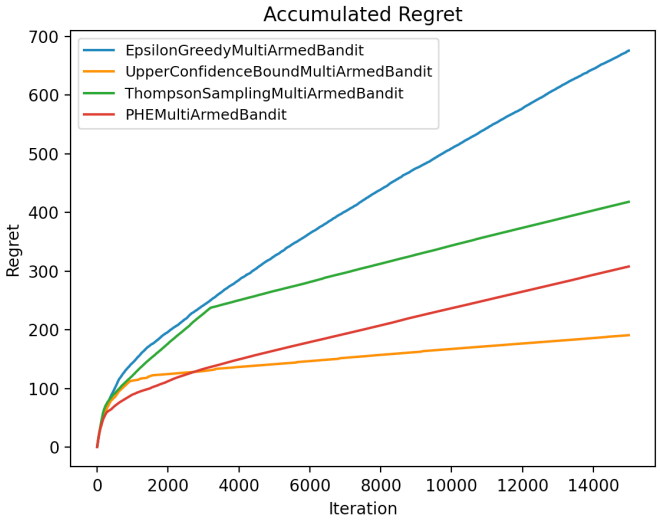


Figure 9: noise = 0.5, K = 25

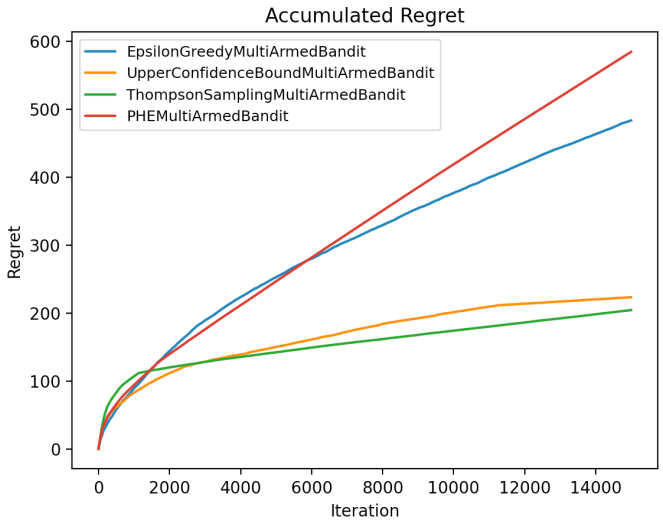


Figure 10: noise = 1.0, K = 25

All algorithms output similar results as the noise remains the same and number of arms increases. This is because with greater number of arms, more complex algorithms tend to do as well, because they all discover arms that output great distribution of values relatively fast. However, all algorithms seem to struggle as the noise increases. This is due to an imbalance between exploration and exploitation. When the noise is larger, it is hard for algorithms to find the true mean of each arm, because the rewards the agent receives tend to vary more, meaning it is harder to draw a conclusion on the true distribution of rewards. Overall, PHE performs the best, leading to almost optimal regret, unless the noise value is large, in which PHE performs poorly compared to other algorithms. Thompson Sampling and UCB have similar outcomes, partly because both algorithms employ similar techniques to find the true distribution of each arm's reward. TS tend to perform a little better than UCB overall. Epsilon Greedy performs the worst in all cases, because it uses random exploration strategy, and does not

II. Contextual Linear Bandit Problem

In Contextual Linear Bandit Problem, each user has an unobservable linear parameter which parameterizes the linear reward function that takes the feature vector as input. At every step, the agent observes and selects an arm based on the feature vectors and receives a reward for picking the arm. Each algorithm implemented for multi-Armed bandit has a linear counterpart, and these algorithms can be applied for the agent to learn the relationship between feature vectors and rewards. The algorithm is described as follows: the agent observes the current user and a set of arms to choose from, $a \in A_t$. The context vector $x_{t,a}$ summarizes information of both the user and arm a. Based on previous observations, the agent chooses an arm $a_t \in A_t$, and receives a scalar reward r_{t,a_t}. The algorithm would improve its strategy and repeat the process.

A. Epsilon-Greedy Contextual Bandit

Similar to Epsilon-Greedy with Multi-Armed Bandit, the algorithm would choose a random arm with probability ϵ, and choose the best arm based on history with probability $1 - \epsilon$. In ϵ -greedy with contextual bandits, context vector $x_{t,a}$ is represented as a vector with its size equal to the feature dimension d. The expected reward of an arm a is linear in its d-dimentional feature $x_{t,a}$, with some unknown coefficient vector θ_a, initialized to $\vec{0}$ with side d. Therefore, the expected reward is computed as follows:

$$\mathbb{E}[r_{t,a}|x_{t,a}] = x_{t,a}^T \theta_a$$

The algorithm would perform this calculation for all arm at time t, and picks the arm that generates the greatest value of $r_{t,a}$.

$$a_t = \arg\max x_{t,a}^T \theta_a$$

In order to learn the values for θ_a, we would need online learning algorithm with ridge regress to update the value for θ_a at each timestep. Let D_a be a design matrix of dimension $m \times d$, in which m is the number of training inputs. Let b_a be the response vector of size m to D_a. Perform ridge regression to the training data to obtain the updated value of θ_a at each timestep:

$$\theta_a = (D_a^T D_a + I_d)^{-1} D_a^t c_a$$

I_d is a $d \times d$ identity matrix and c_a is the components independent conditioned to corresponding rows in D_a. The equation above can be simplified.

$$\theta_a = A_a^{-1} b_a$$

Where A_a is a matrix of size $d \times d$ initialized to λI, in which λ is a tunable scalar hyperparameter, and I_d is a $d \times d$ identity matrix. b_a is a vector of size d, initialized to all 0s at first. After an arm is pulled, A_a will add the dot product between the feature vector and its transposed vector.

$$A_a = A_a + x_{t,a} x_{t,a}^T$$

Vector b_a will add the product of the feature vector with the reward.

$$b_a = b_a + x_{t,a} r_a$$

And θ_a can be calculated to be used for arm selection again in the next step. The complete algorithm is shown in Algorithm 5.

Algorithm 5 Linear ϵ-greedy

0: Inputs: d = feature dimension, $\lambda = 0.1$, $\epsilon = (\mathrm{n}+1)^{\frac{-1}{3}}$
1: Initialize: $A_{d\times d} = \lambda I$, $b = 0_d$, $\theta = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: **if** $\mathcal{B}(1,\epsilon)$ is 1 **then**
4: choose random arm to play
5: **else**
6: Play arm $a := \arg\max x_{t,a}^T \theta_a$ and observe r_a
7: Update:
8: $A_a = A_a + x_{t,a}x_{t,a}^T$
9: $b_a = b_a + x_{t,a}r_a$
10: $\theta_a = A_a^{-1}b_a$
11: **end for**

B. **Upper Confidence Bound Contextual Bandit (LinUCB)**
LinUCB uses the same idea as UCB. The expected reward for arm a given the feature vector is computed as follows:

$$\mathbb{E}[r_{t,a}|x_{t,a}] = x_{t,a}^T\theta_a$$

The variance of the expected reward is computed as follows:

$$x_{t,a}^T\theta_a = x_{t,a}^T A_a^{-1}x_{t,a}$$

This means that $\sqrt{x_{t,a}^T A_a^{-1}x_{t,a}}$ would become the standard deviation of the model. The agent picks and arm based on the maximum value of UCB_{score}, which is computed as follows:

$$UCB_{score} = x_{t,a}^T\theta_a + \alpha\sqrt{x_{t,a}^T A_a^{-1}x_{t,a}}$$

$x_{t,a}^T\theta_a$ computes the mean of arm a at time t, and $\alpha\sqrt{x_{t,a}^T A_a^{-1}x_{t,a}}$ computes the confidence interval with hyperparameter weight α. α is often computed as $\alpha = 1 + \sqrt{\ln\frac{\left(\frac{2}{\delta}\right)}{2}}$, in which δ is a integer $\delta > 0$.

Algorithm 6 LinUCB

0: Inputs: d = feature dimension, $\lambda = 0.1$, $\alpha = 0.25$
1: Initialize: $A_{d\times d} = \lambda I$, $b = 0_d$, $\theta = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: Play all arms once
4: Play arm $a := \arg\max x_{t,a}^T\theta_a + \alpha\sqrt{x_{t,a}^T A_a^{-1}x_{t,a}}$
5: Update:
6: $A_a = A_a + x_{t,a}x_{t,a}^T$
7: $b_a = b_a + x_{t,a}r_a$
8: $\theta_a = A_a^{-1}b_a$
9: **end for**

C. **Thompson Sampling Contextual Bandit (LinTS)**
LinTS follows the same principle as Thompson Sampling for Multi-armed bandits. At each timestep, the algorithm will sample from the prior distribution, and updates the mean and standard deviation to the posterior: $\mathcal{P}(\theta|\lambda) = \mathcal{N}(0,\lambda I)$. We also assume the standard deviation of the reward distribution is known, $\sigma = Gaussian\ noise$, so we will sample from prior and posterior using Gaussian distributions,

Algorithm 7 LinTS

0:Inputs: d = feature dimension,$\lambda = 0.1, \sigma = Gaussian\ noise$
1: Initialize: $A_{d\times d} = \lambda I$, $b = 0_d$, $\theta = 0_d$
2: **for** t = 1, 2, 3 …, T **do**
3: Play arm $a := \arg\max x_{t,a}^T\theta_a$
4: Update:
5: $A_a = A_a + \frac{1}{\sigma^2}x_{t,a}x_{t,a}^T$
6: $b_a = b_a + \frac{1}{\sigma^2}x_{t,a}r_a$
7: $\theta_a = \mathcal{N}(A_a^{-1}b_a, A_a^{-1})$
8: **end for**

D. Contextual Bandit Analysis
This portion of analysis consists of three algorithms: Linear ϵ-greedy, LinUCB and LinTS. The following algorithm hyperparameters remain the same throughout all experiment runs:

ϵ-greedy: $\lambda = 0.1$, $\epsilon = (\mathrm{n}+1)^{\frac{-1}{3}}$, n = time
LinUCB: $\lambda = 0.1$, $\alpha = 0.25$
LinTS: $\lambda = 0.1$
*for LinTS, σ will change with the Gaussian noise: $\sigma = gaussian\ noise$

The following environment hyperparameters remain unchanged for Trial 1 through 5.

Testing iterations: 15,000 steps
Number of arms: 25
Number of users: 10

Trial 1-3 Constant noise = 0.1, Context dim = 3, 25, 50
Figure 11-13

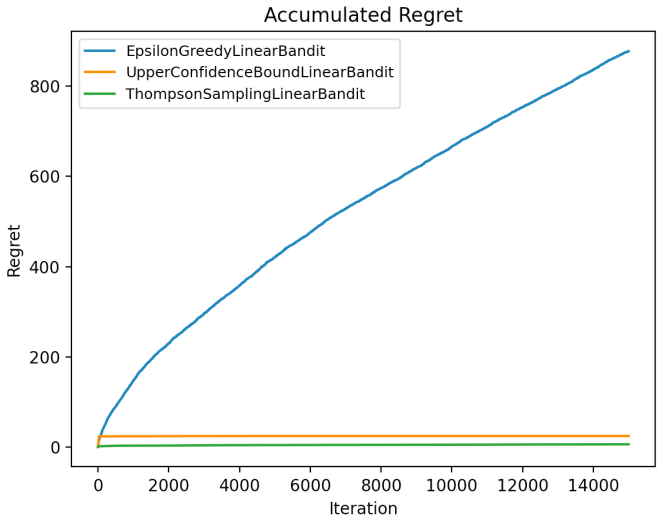


Figure 11a: noise = 0.1, Context dim = 3

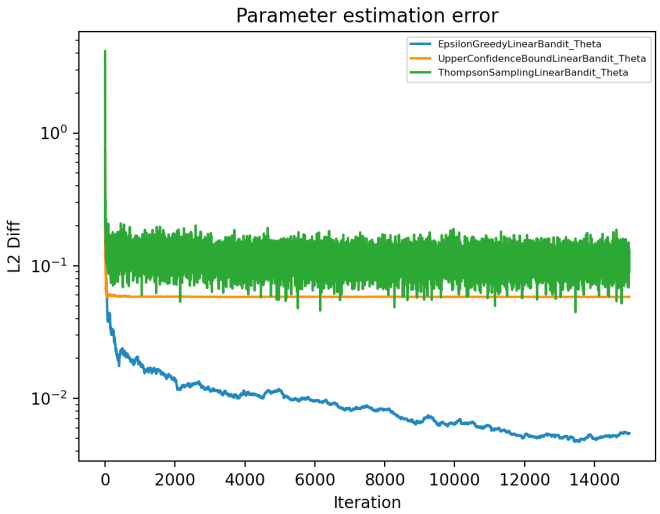


Figure 11b: noise = 0.1, Context dim = 3

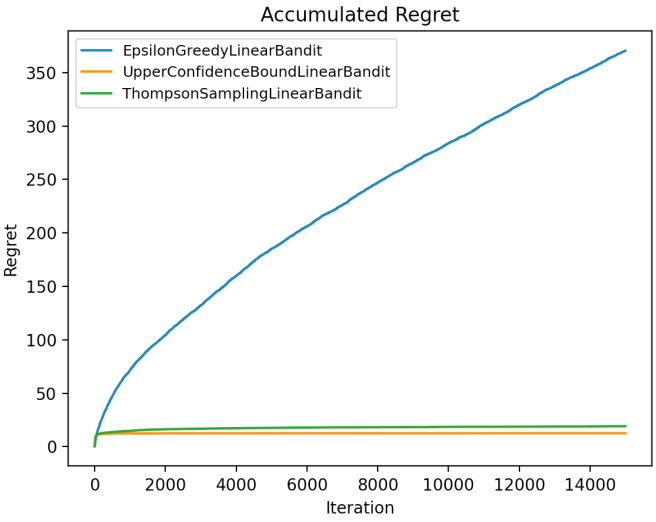


Figure 12a: noise = 0.1, Context dim = 25

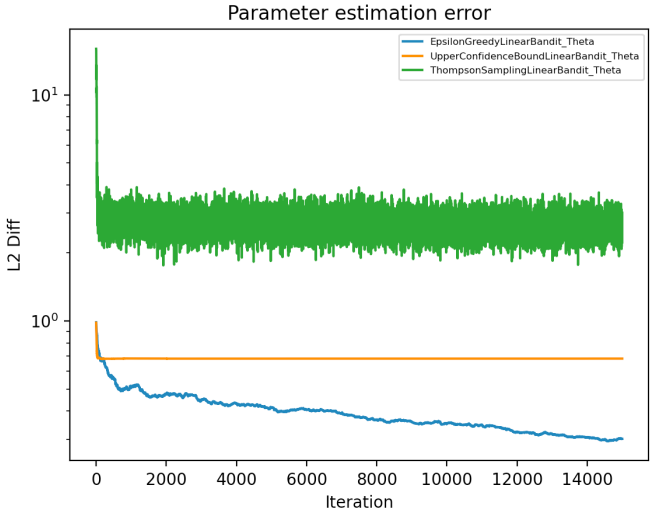


Figure 12b: noise = 0.1, Context dim = 25

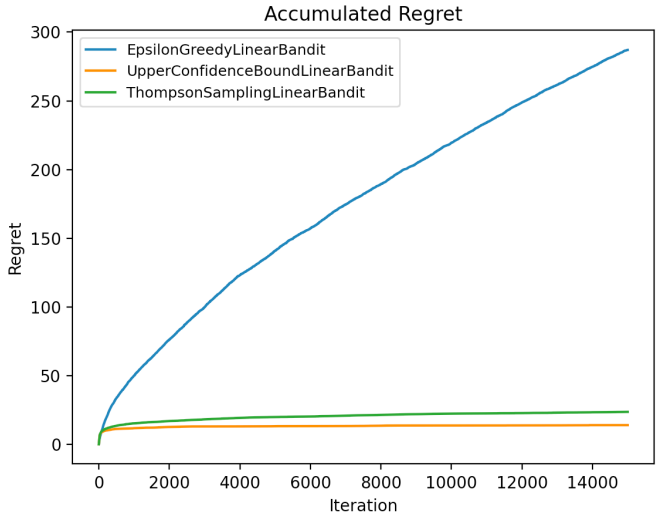


Figure 13a: noise = 0.1, Context dim = 50

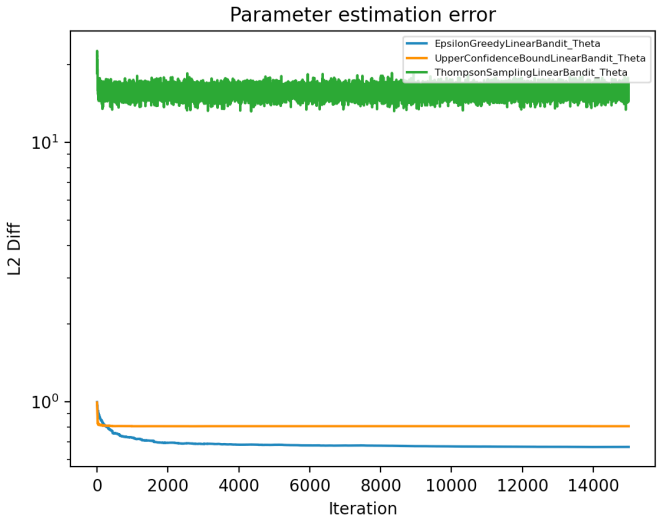


Figure 13b: noise = 0.1, Context dim = 50

	Noise	**Context dimension**	**Action set size**
Trial 1	0.1	**5**	Full info
Trial 2	0.1	**25**	Full info
Trial 3	0.1	**50**	Full info

	Dim = 5	**Dim = 25**	**Dim = 50**
Algorithm	**Trial 1 regret**	**Trial 2 regret**	**Trial 3 regret**
Lin ϵ-greedy	877.41	370.33	287.05
LinUCB	25.13	12.58	14.15
LinTS	6.69	19.11	23.79

For Linear Epsilon Greedy, higher context dimensions leads to lower total regret. This is because the mapping between the feature vectors and rewards contains more information as the size of the feature vectors becomes larger, making θ_a more accurate. Another reason is that since the Gaussian noise is relatively a smaller value $\sigma = 0.1$, θ_a would be good at generalizing the true Gaussian distribution of each arm when the context dimension is larger. The regret for LinUCB is better when the size of context dimension is the same as the number of arms, but additional size increase in the

context vector does not provide a better regret result. The parameter estimation error for LinUCB also remained consistent in all three trials, meaning that the size of the feature vector dimension does not yield a significant change to the overall algorithm performance. The regret for LinTS and its parameter estimation error increased as the size of the context vectors increased. This is because with more dimensions, TS needs more time to ultimately converge to an ideal distribution θ_a, because more exploration is needed for runs with higher context vector dimensions.

Trial 4-5: Constant Context dim = 25, noise = 0.05, 0.5
Figure 14-15

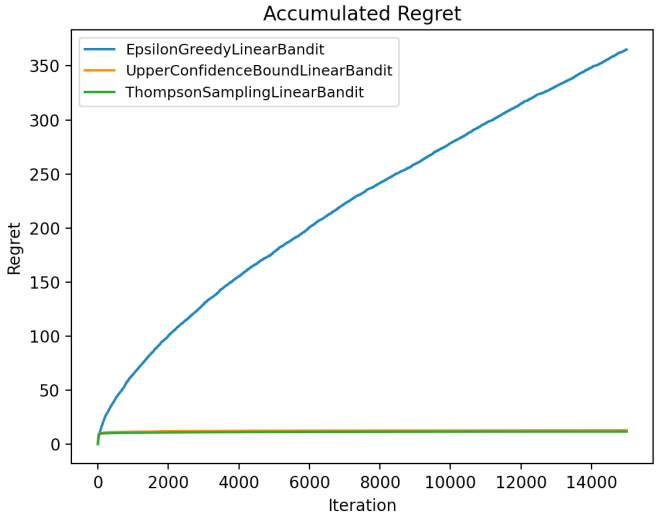


Figure 14a: noise = 0.05, Context dim = 25

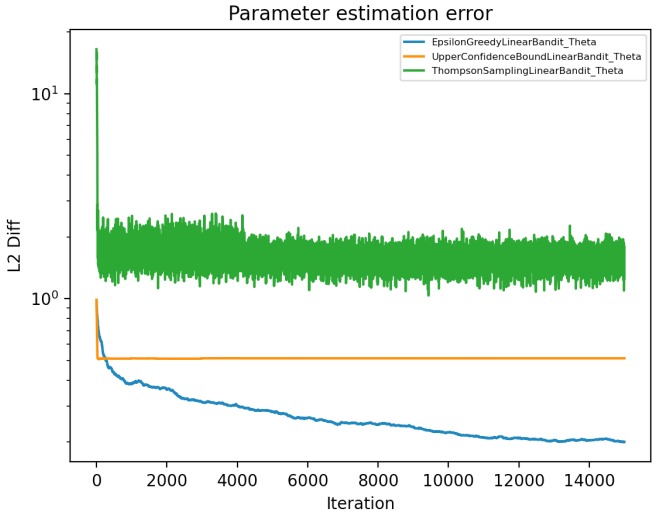


Figure 14b: noise = 0.05, Context dim = 25

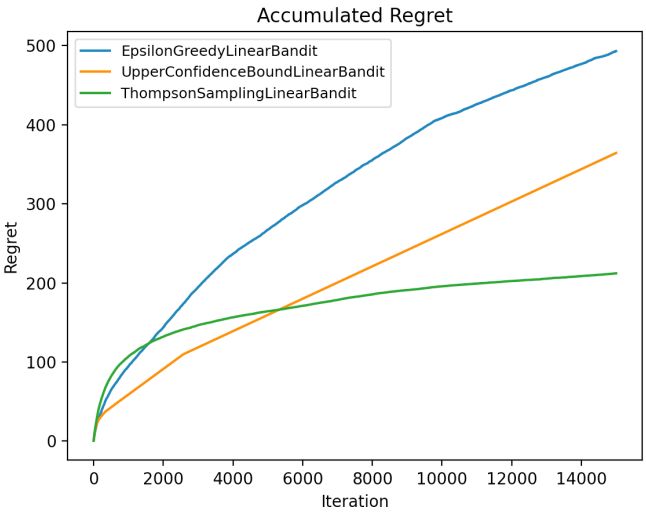


Figure 15a: noise = 0.5, Context dim = 25

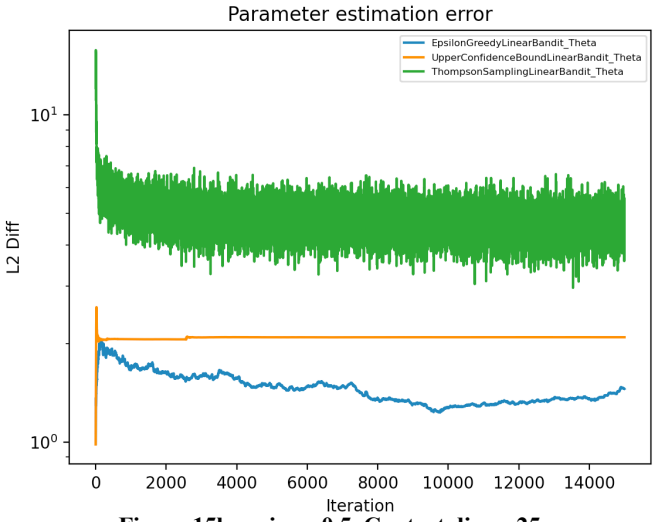


Figure 15b: noise = 0.5, Context dim = 25

	Noise	**Context dimension**	**Action set size**
Trial 4	**0.05**	25	Full info
Trial 5	**0.5**	25	Full info

	Noise = 0.05	**Noise = 0.5**
Algorithm	**Trial 4 regret**	**Trial 5 regret**
Lin ϵ-greedy	364.98	493.04
LinUCB	13.14	364.26
LinTS	11.88	212.07

All algorithms perform significantly worse as the Gaussian noise level increases. This is more apparent in LinUCB and LinTS, because both algorithms try to estimate the reward distribution of each arm. Thus, if the Gaussian noise level is high for all arms, then it is more likely for each arm's gaussian distribution to overlap, so that it is harder for both LinUCB and LinTS to estimate the true reward distribution.

Trial 6 – 9 experiments with different size of the action set and see how it influences each algorithm's performance. The following environment hyperparameters are used:

Testing iterations: 15,000 steps
Context Dimension size: 25
Number of arms: 100
Number of users: 10
Gaussian noise: 0.1

Constant K = 100, Constant noise = 0.1, Context dim = 25, poolArticleSize = 5, 10, 20
Figure 16-18

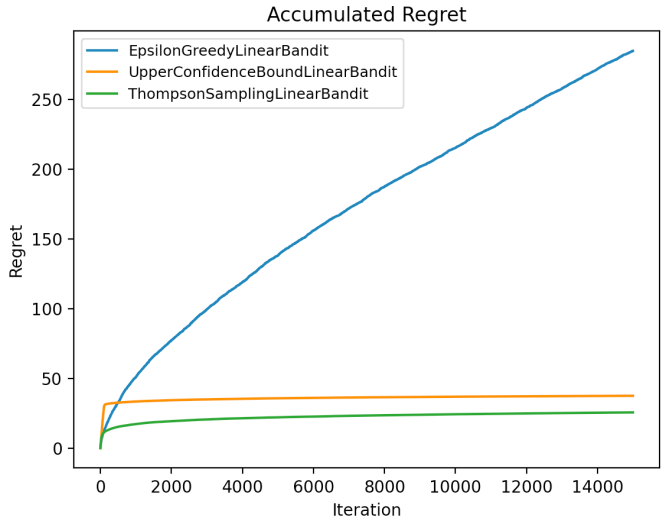


Figure 16a: noise = 0.1, Context dim = 25, poolArticleSize = 10

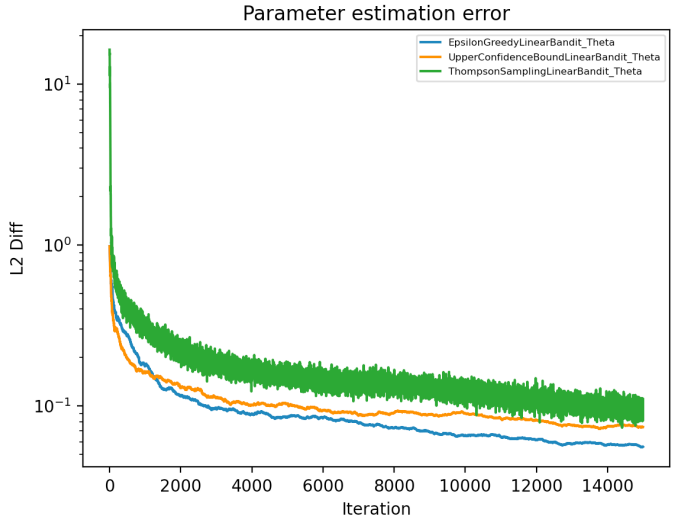


Figure 16b: noise = 0.1, Context dim = 25, poolArticleSize = 10

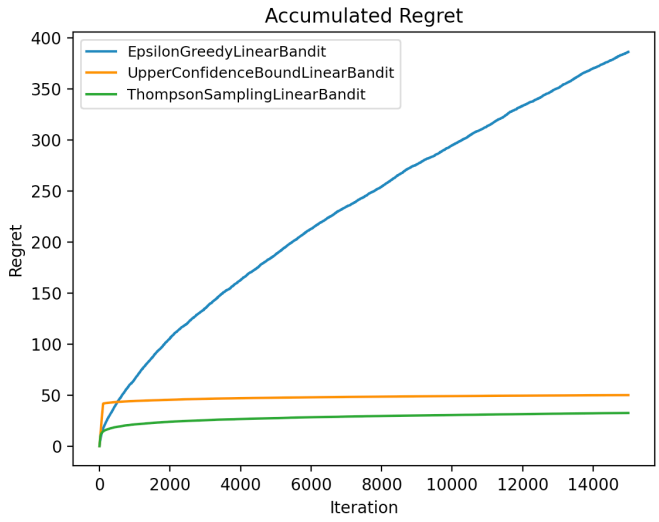


Figure 17a: noise = 0.1, Context dim = 25, poolArticleSize= 40

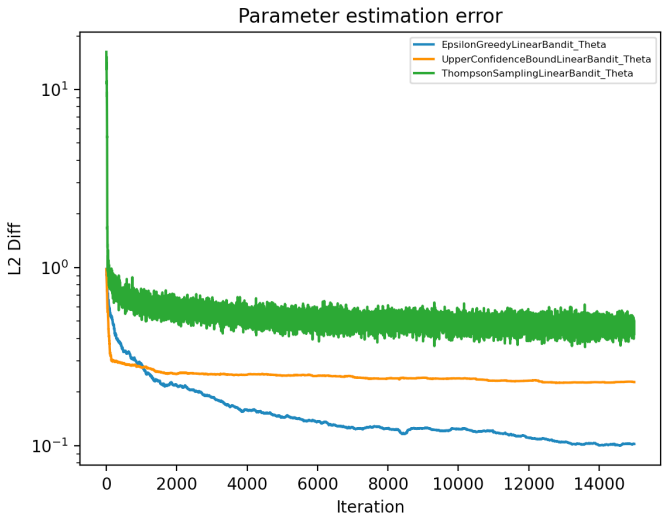


Figure 17b: noise = 0.1, Context dim = 25, poolArticleSize= 40

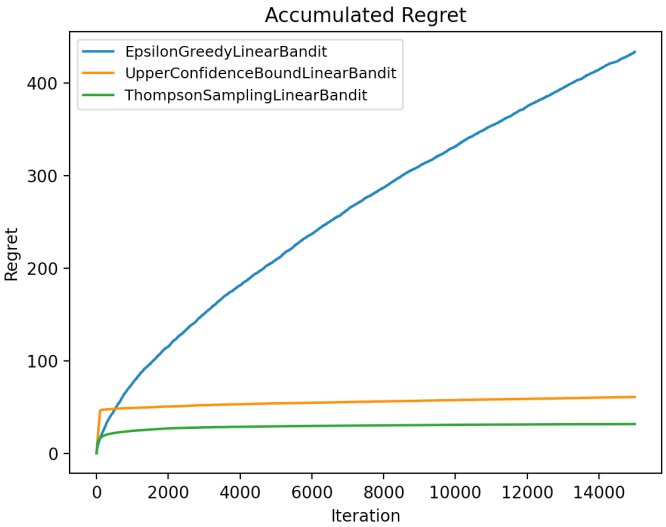


Figure 18a: noise = 0.1, Context dim = 25, poolArticleSize= 85

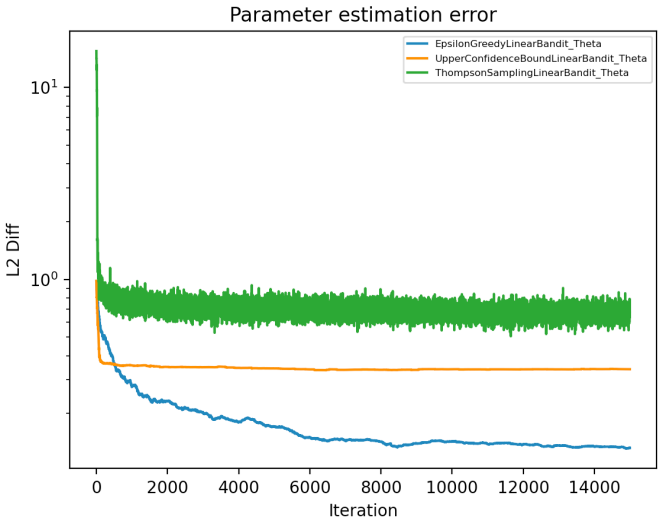


Figure 18b: noise = 0.1, Context dim = 25, poolArticleSize= 85

	Noise	Context dimension	K	Action set size
Trial 6	0.1	25	100	**10**
Trial 7	0.1	25	100	**40**
Trial 8	0.1	25	100	**85**

	Actionset=10	Actionset=40	Actionset=85
Algorithm	**Trial 6 regret**	**Trial 7 regret**	**Trial 8 regret**
Lin ϵ-greedy	284.94	386.19	433.51
LinUCB	37.79	50.32	61.29
LinTS	25.87	32.80	32.02

The environment randomly samples a size of "Actionset" in each timestep, which is a subset of articles as the action set. This means that at each timestep, the learner does not have all available arms to pick. Therefore, it is possible that the learner does not get to pick the best arm, simply because the sampled arm pool does not contain such arm at the current timestep.

Linear Epsilon Greedy performs worse as the size of the action set increase. When the action set is small, there can only be a few arms to choose from, meaning the algorithm has less room to explore. When the action set is larger, there are a lot more arms to explore, and more time is needed to sufficiently explore all the arms and estimate their true reward distributions. Linear UCB also follows a similar trend. When the number of actions to choose from is small, the algorithm would not have to estimate a lot of confidence intervals, but it takes a lot more time to estimate the confidence intervals for every arm is the action set is bigger. Linear Thompson Sampling was not greatly affected by the increase in the size of the action set. This is because the technique of exploration and exploitation is embedded in the TS algorithm, meaning the algorithm would adjust its exploration and exploitation rate based on the side of the action set.

III. INFLUENCE OF THE SHAPE OF ACTION SET

This section compares performance between linear bandit algorithms and multi-armed bandit algorithms. In order to establish a fair comparison, the following hyperparameters relationships are assumed:

PoolArticleSize = None
Number of articles = context dimension = K

Experiments with different K values and different shape of the action set are conducted. There are two types of shapes for the action set: basis vector, or random. Basis vector constrains the articles feature vectors to basis vectors like

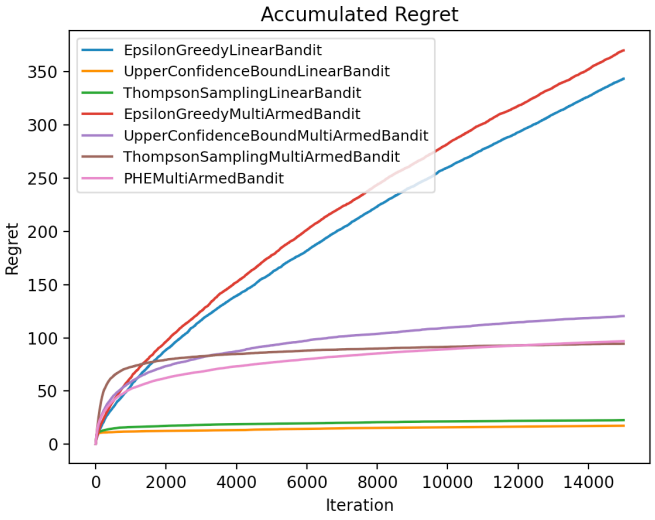
$$e_0 = [1,0,0,\dots,0]$$
$$e_1 = [0,1,0,\dots,0]$$

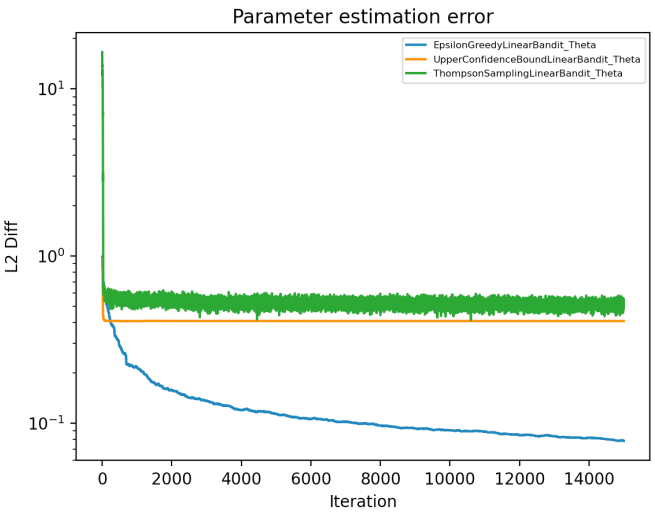
Therefore, the feature vectors of the articles will be orthogonal, meaning that observation about the reward of one article brings no information about the reward of another article. Random shape of the action set means that feature vectors will be randomly sampled, and the correlation between feature vectors to rewards could be found using linear bandit algorithms.

The following experiments are conducted:

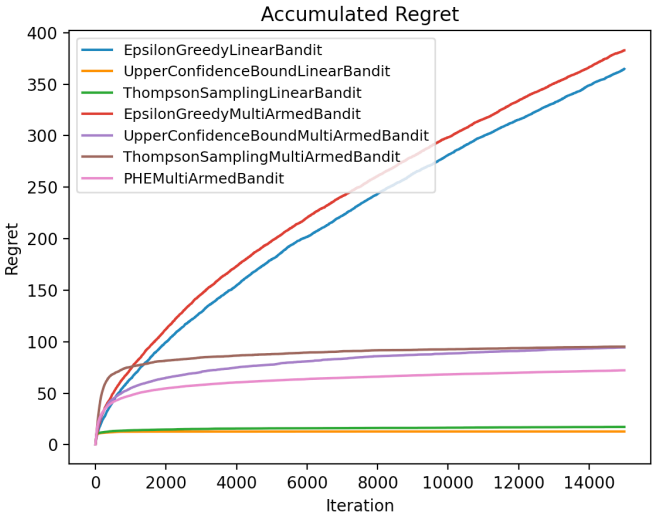
Trials	Shape	K	noise	steps	Num users
1	**Basis vector**	**25**	0.1	15,000	10
2	**Random**	**25**	0.1	15,000	10
3	**Basis vector**	**100**	0.1	15,000	10
4	**Random**	**100**	0.1	15,000	10

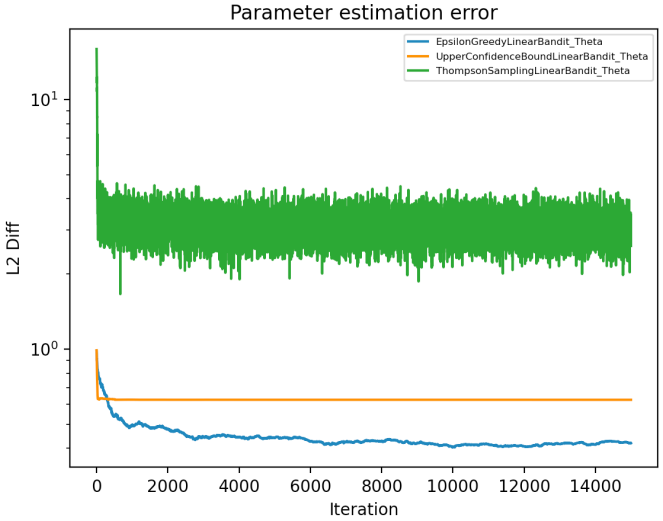
Trial 1 (basis vector action set)





Trial 2 (random shaped action set)





Actionset = "random"

Algorithm	**"basis vector" regret**	**"random" regret**
ϵ-greedy	370.17	382.92
UCB	120.48	94.43
TS	22.71	95.21
PHE	94.48	72.23
Lin ϵ-greedy	343.59	364.78
LinUCB	17.44	12.82
LinTS	22.71	17.27

For linear bandits, the accumulated regret under random shaped action sets is smaller. The principal relationships between contextual bandit problems and multi-armed bandit problems are the contextual bandit problem becomes multi-armed bandit problem when the action set A_t is unchanged and contains K actions for all t, and the user, or the context, is the same for all t. This means that multi-armed bandit problems are just contextual bandit problems without the context. Mathematically, a context-free bandit would have its feature $x_{t,a} = 1$, and $\theta_a = \mu_a$, which correlates to the "basis vector" shaped action set. In a contextual linear bandit, the expected reward given the feature vector is computed as follows:

$$\mathbb{E}[r_{t,a}|x_{t,a}] = x_{t,a}^T \theta_a$$

For a feature-free bandit with $x_{t,a} = 1$, it can be written in the same format as the contextual bandit:

$$\mathbb{E}[r_{t,a}|x_{t,a}] = 1^T \mu_a$$

Based on these observations, we can compute the regret bound for both linear bandit algorithms and multi-armed bandit algorithms.

MAB:

$$R(T) \geq \frac{KlogT}{\min_{a_i \neq a*} KL(p_{a^*}, p_{a_i})}$$

lower regret bound: $R(T) = \Omega(\gamma\sqrt{TK})$

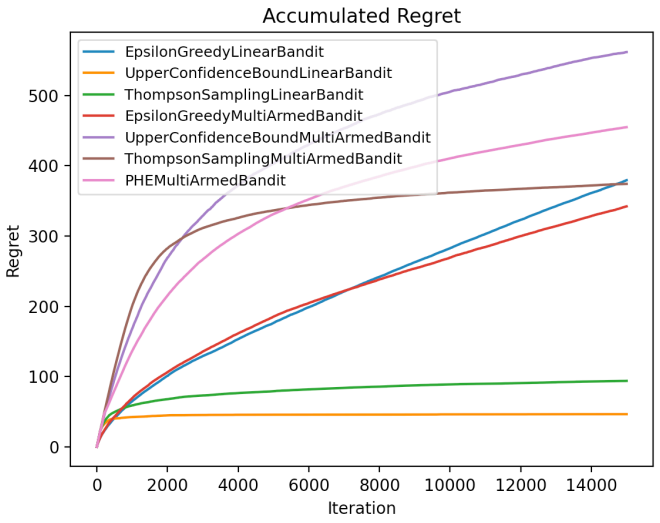
Linear Bandits:

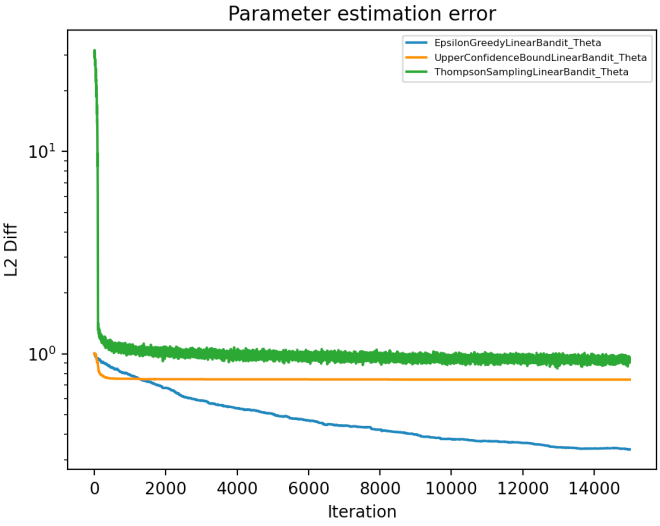
$$R(T) = O(\sqrt{Td}\log\left(\lambda + \frac{T}{d}\right))$$

lower regret bound: $R(T) = \Omega(\gamma\sqrt{TK})$, when $d \leq \sqrt{T}$

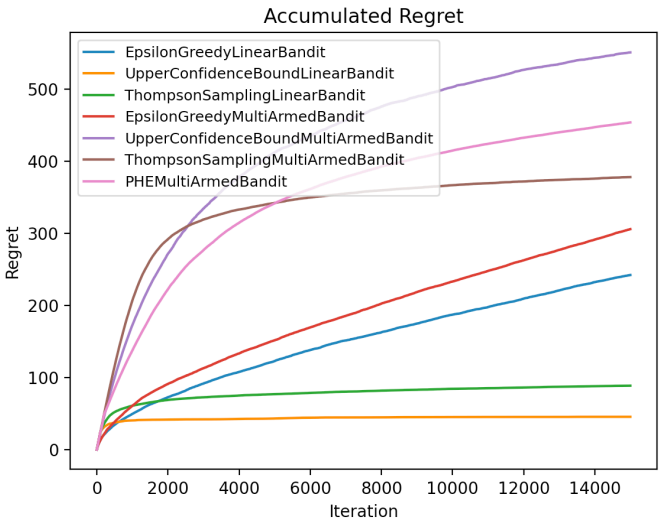
Based on these computations, it is clear that linear bandits would outperform non-linear bandits. And my experimental results confirm that all linear bandit algorithms outperform multi-armed bandit algorithms.

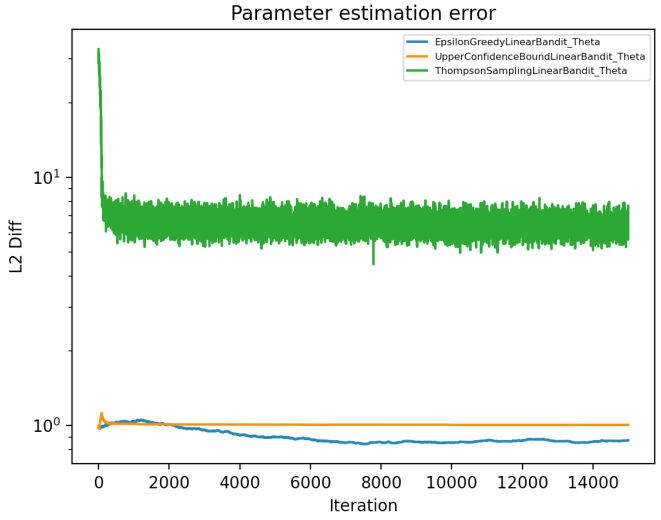
Trial 3





Trial 4





Algorithm	**"basis vector" regret**	**"random" regret**
ϵ-greedy	342.10	305.74
UCB	561.60	550.67
TS	374.11	377.86
PHE	454.75	453.63
Lin ϵ-greedy	379.44	242.16
LinUCB	46.65	45.63
LinTS	94.00	88.74

All algorithms perform poorly when K = 100. This is because due to the large size. However, linear bandit algorithms still completely outperform multi-armed bandit algorithms, and that is because learning the mapping between linear contexts could greatly improve the arm selection process.

Appendix A

Epsilon Greedy

```
    def updateParameters(self, articlePicked_id, click):
        self.UserArmMean[articlePicked_id] =
(self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + click) / \
(self.UserArmTrials[articlePicked_id]+1)
        self.UserArmTrials[articlePicked_id] += 1

        self.time += 1

    def decide(self, pool_articles):
        if self.epsilon is None:
            explore = np.random.binomial(1, (self.time+1)**(-1.0/3))
            # explore = np.random.binomial(1, np.min([1, self.d/self.time]))
        else:
            explore = np.random.binomial(1, self.epsilon)
        if explore == 1:
            # print("EpsilonGreedy: explore")
            articlePicked = np.random.choice(pool_articles)
        else:
            # print("EpsilonGreedy: greedy")
            maxPTA = float('-inf')
            articlePicked = None

            for article in pool_articles:
                article_pta = self.UserArmMean[article.id]
                # pick article with highest Prob
                if maxPTA < article_pta:
                    articlePicked = article
                    maxPTA = article_pta

        return articlePicked
```

Upper Confidence Bound

```
   def updateParameters(self, articlePicked_id, click):
        self.UserArmMean[articlePicked_id] =
(self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + click) / \
(self.UserArmTrials[articlePicked_id]+1)
        self.UserArmTrials[articlePicked_id] += 1

        self.time += 1
   def decide(self, pool_articles):
        maxValue = float('-inf')
        articlePicked = None
        for article in pool_articles:
            # play all the arms once first
            if self.UserArmTrials[article.id] == 0:
                return article
            article_value = self.UserArmMean[article.id] + (self.alpha * np.sqrt((2 * \
np.log(self.time)) / self.UserArmTrials[article.id]))
            if maxValue < article_value:
                articlePicked = article
                maxValue = article_value
        return articlePicked
```

Thompson Sampling

```
    def updateParameters(self, articlePicked_id, click):
        self.UserArmMean[articlePicked_id] =
(self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + click) /
(self.UserArmTrials[articlePicked_id]+1)
        self.UserArmTrials[articlePicked_id] += 1

        self.time += 1
    def decide(self, pool_articles):
        maxPTA = float('-inf')
        articlePicked = None

        for article in pool_articles:
            if self.UserArmTrials[article.id] == 0:
                return article
            article_pta = np.random.normal(self.UserArmMean[article.id], 1 /
self.UserArmTrials[article.id])
            # pick article with highest Prob
            if maxPTA < article_pta:
                articlePicked = article
                maxPTA = article_pta

        return articlePicked
```

Perturbed History Exploration

```
    def updateParameters(self, articlePicked_id, click):
        self.UserArmMean[articlePicked_id] =
(self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + click) /
(self.UserArmTrials[articlePicked_id]+1)
        self.UserArmTrials[articlePicked_id] += 1

        self.T[articlePicked_id] += 1
        self.V[articlePicked_id] += click

        self.time += 1
    def decide(self, pool_articles):

        maxPTA = float('-inf')
        articlePicked = None

        for article in pool_articles:
            s = self.UserArmTrials[article.id]
            if self.T[article.id] > 0:
                U = np.random.binomial(ceil(self.a * s), 0.5)
                mu = (self.V[article.id] + U) / ((self.a + 1) * s)
            else:
                mu = np.inf

            article_pta = mu
            if maxPTA < article_pta:
                articlePicked = article
                maxPTA = article_pta

        return articlePicked
```

Linear Epsilon Greedy

```
    def updateParameters(self, articlePicked_FeatureVector, click):
        self.A += np.outer(articlePicked_FeatureVector, articlePicked_FeatureVector)
        self.b += articlePicked_FeatureVector * click
        self.AInv = np.linalg.inv(self.A)
        self.UserTheta = np.dot(self.AInv, self.b)
        self.time += 1
    def decide(self, pool_articles):
        if self.epsilon is None:
            explore = np.random.binomial(1, (self.time+1)**(-1.0/3))
        else:
            explore = np.random.binomial(1, self.epsilon)
        if explore == 1:
            # print("EpsilonGreedy: explore")
            articlePicked = np.random.choice(pool_articles)
        else:
            # print("EpsilonGreedy: greedy")
            maxPTA = float('-inf')
            articlePicked = None

            for article in pool_articles:
                article_pta = np.dot(self.UserTheta, article.featureVector)
                # pick article with highest Prob
                if maxPTA < article_pta:
                    articlePicked = article
                    maxPTA = article_pta
        return articlePicked
```

Linear Upper Confidence Bound

```
    def updateParameters(self, articlePicked_FeatureVector, click):
        # expected value(r_a_t) = E[r(t,a)|x(t|a)]
        self.A += np.outer(articlePicked_FeatureVector, articlePicked_FeatureVector)
        self.b += articlePicked_FeatureVector * click
        self.AInv = np.linalg.inv(self.A)
        self.UserTheta = np.dot(self.AInv, self.b)

    def decide(self, pool_articles):

        maxPTA = float('-inf')
        articlePicked = None

        for article in pool_articles:
            # play all the arms once first
            if article not in self.play_dict:
                articlePicked = article
                self.play_dict[article] = 1
                break
            else:
                article_pta = np.dot(self.UserTheta, article.featureVector) + (self.alpha * \
np.sqrt(np.dot(np.dot(np.transpose(article.featureVector), self.AInv), article.featureVector)))

                if maxPTA < article_pta:
                    articlePicked = article
                    maxPTA = article_pta

        return articlePicked
```

Linear Thompson Sampling

```
    def updateParameters(self, articlePicked_FeatureVector, click):
        self.A += (1/(self.sigma**2)) * np.outer(articlePicked_FeatureVector,
articlePicked_FeatureVector)
        self.b += (1/(self.sigma**2)) * (articlePicked_FeatureVector * click)
        self.AInv = np.linalg.inv(self.A)
        self.UserTheta = np.random.multivariate_normal(np.dot(self.AInv, self.b), self.AInv)
        self.time += 1
    def decide(self, pool_articles):

        maxPTA = float('-inf')
        articlePicked = None

        for article in pool_articles:
            article_pta = np.dot(self.UserTheta, article.featureVector)

            if maxPTA < article_pta:
                articlePicked = article
                maxPTA = article_pta

        return articlePicked
```