Imperial College London

Imperial College London
Derpartment of Mathematics

# Mixture Model Bandit

Jiaxu Li

CID: 01378821

Supervised by Ciara Pike-Burke

12th December 2021

Submitted in partial fulfilment of the requirements for the MSc in Statistics of Imperial College London

The work contained in this thesis is my own work unless otherwise stated.

Signed: Jiaxu Li Date: 20th August 2021

# Abstract

This main topic of this project is mixture model bandits. This project is mainly a simulation study that aims to compare the performances of different algorithms under different conditions with the mixture model assumptions. All the bandit algorithms seek to maximize the cumulative rewards in each term or play. Some algorithms are UCB-based, meaning they exploit the upper confidence bounds as a criterion of choosing arms. Other algorithms include Thompson sampling, which places a prior on the parameters of the bandit environment, and also epsilon-greedy algorithm, which uses explore-and-then-commit principle.

A main tool used here is the EM algorithm, which is used to take into account the mixture model structure and calculate estimates of the mixture distributions. A main question here is whether the implementation of EM algorithm in bandit algorithms would lead to a significant improvement in the performance. EM algorithm takes more computational efforts but brings us with a more accurate estimation of the mixture model parameters.

In this project, simulations are firstly done for UCB and UCB-V with and without EM, Thompson sampling, UCB-Normal algorithms under the Gaussian setting, i.e., the rewards of arms are mixtures of Gaussians. Then, Beta reward distributions are also studied to compare UCB-V with and without EM.

# Acknowledgements

I would like to thank Dr Ciara Pike-Burke for her expert advice and encouragement during the completion of this project.

# Table of Contents



The sections marked with (*) are mostly my original work.

# 1. Introduction

## 1.1. Backgrounds

In real life, a two-armed bandit is a usually a (gambling) machine of which humans could decide to pull either the left or the right arm, each giving a reward. However, the player has generally no information at all about the distribution of the reward from each different machine. Intuitively, the human player wants to maximize his cumulative rewards throughout his plays. It was named a two-armed bandit in paying tribute to the one-armed bandit, an archaic name for a lever-operated coin machine.

Bandit problems were initially introduced by William R. Thompson. He was interested in medical trials and the humanity in running them. The name can be traced back to 1950s, when Frederick Mosteller and Robert Bush did experiments on animal learning through trials on mice and subsequently on humans. In the experiment, the mice were placed in at the bottom of a T-shaped box and had to choose which way to go to find food. The same experiment was then repeated on the mice for multiple times to study the learning process of the mice and see whether the mice could maximize the cumulative times of finding food at the end of the passage of their choices.

The importance of studying bandit problems is becoming increasingly important these days. All human have to make decisions with uncertainty, and bandit problems can provide an applicable model for this problem.

In a clinical trial design, researchers have used bandit models to generate ideas for their works for nearly 80 years. Adaptive trial design is becoming more and more popular and is recommended by the US Food and Drug Administration because not doing so may result in effective drugs being withheld until a beneficial effect has long been demonstrated. Large technology companies have used bandit algorithms in applications such as news recommendations, dynamic pricing, and ad placement are used. A Bandit algorithm had also been implemented in Monte Carlo Tree Search, an algorithm made famous by the recent feat of Alpha-Go (Lattimore and Szepesvári, 2020).

## 1.2. The Language of Bandits

A bandit problem is a sequential game between a learner and an environment. The game is played over $n$ rounds, where $n$ is a positive natural number called the horizon.

In each round $t \in [n]$, the learner first chooses an action $A_t$ from a given set $\mathcal{A}$, and the environment then reveals a reward $X_t \in \mathbb{R}$.

The learner cannot predict the future when deciding their actions, which means that $A_t$ should only be dependent on the history $H_{t-1} = (A_1, X_1, \ldots, A_{t-1}, X_{t-1})$ . A policy is a mapping from histories to actions: A learner accept a policy to have interactions with an environment. An environment is a mapping from history sequences ending in actions to rewards. Both the learner and the environment may randomize their decisions. The most common goal of the learner is to choose actions that lead to the largest possible cumulative reward over all $n$ rounds, $\sum_{t=1}^{n} X_t$.

The main challenge in bandit problems is that the environment is unknown to the learner. All the information that a learner has access to is that the true environment lies in some set $\mathcal{E}$ called the environment class.

There are several measures to evaluate a learner. Most of the efforts in designing a successful policy are dedicated to understand the regret. There are multiple ways to define this quantity. The most general definition is firstly given here.

## 1.3. Stochastic Bandits, Regrets, and Mixture Model Assumptions

A stochastic bandit is a specific kind of bandit problem, which contains a collection of distributions $\nu = (P_a : a \in \mathcal{A})$, where $\mathcal{A}$ is the set of available actions. The learner and environment interact sequentially over $n$ rounds. In each round $t \in \{1, \ldots, n\}$, the learner chooses an action $A_t \in \mathcal{A}$, which is fed to the environment. The environment then samples a reward $X_t \in \mathbb{R}$ from distribution $P_{A_t}$ and reveals $X_t$ to the learner. The interaction between the learner (or policy) and environment induces a probability measure on the sequence of outcomes $A_1, X_1, A_2, X_2, \ldots, A_n, X_n$. In most cases, the horizon $n$ is finite, but sometimes it's allowed that the interaction continues infinitely ($n = \infty$). The sequence of actions and rewards should satisfy the following assumptions (Lattimore and Szepesvári, 2020):

(a) The conditional distribution of reward $X_t$ given $A_1, X_1, \ldots, A_{t-1}, X_{t-1}, A_t$ is $P_{A_t}$, which captures the intuition that the environment samples $X_t$ from $P_{A_t}$ in round $t$.

(b) The conditional law of action $A_t$ given $A_1, X_1, \ldots, A_{t-1}, X_{t-1}$ is $\pi_t(\cdot \mid A_1, X_1, \ldots A_{t-1}, X_{t-1})$, where $\pi_1, \pi_2, \ldots$ is a sequence of probability kernels that characterize the learner. The most important assumption is that the learner cannot use the future observations in current decisions.

The learner's goal is to maximize the total reward $S_n = \sum_{t=1}^{n} X_t$, which is a random variable that depends on both the actions of the learner and the rewards sampled from

the environment.

In the previous section, we have defined the regret in a general term as being the deficiency suffered by the learner relative to the optimal policy. Let $\nu = (P_a : a \in \mathcal{A})$ be a stochastic bandit and define

$$\mu_a(\nu) = \int_{-\infty}^{\infty} x dP_a(x)$$

Then let $\mu^*(\nu) = \max_{a \in \mathcal{A}} \mu_a(\nu)$ be the largest mean of all the arms.

**Definition 1.** *(Lattimore and Szepesvári, 2020) The regret of policy $\pi$ on bandit instance $\nu$ is*

$$R_n(\pi, \nu) = n\mu^*(\nu) - \mathbb{E}\left[\sum_{t=1}^{n} X_t\right],$$

where the expectation is taken with respect to the probability measure on outcomes induced by the interaction of $\pi$ and $\nu$. Minimizing the regret is at the same time maximizing the expectation of $S_n$.

In a bandit setting, when measuring the risk, we can either have the random regret or pseudo-regret defined by

$$\hat{R}_n = n\mu^* - \sum_{t=1}^{n} X_t, \qquad \text{(random regret)}$$

$$\bar{R}_n = n\mu^* - \sum_{t=1}^{n} \mu_{A_t}. \qquad \text{(pseudo-regret)} \tag{1.1}$$

In this project and all the plots presented afterwards, pseudo-regrets are used because while $\hat{R}_n$ is influenced by the noise $X_t - \mu_{A_t}$ in the rewards, the pseudo-regret sieves this out by expectation, thus making it a better measure of the 'skill' of a policy.

### 1.3.1. Mixture Model Assumptions

In this project, we focus on a particular bandit environment where the reward of each arm follows a mixture of independent distributions. Suppose $X_t$ is the reward of an arm of the stochastic bandits defined above, then we can write the distribution of $X_t$ as:

$$f_X(x) = \sum_{i=1}^{m} \pi_i g_i(x),$$

where $g_i(x)$ for $i = 1, \ldots, m$ such that $m \geq 2$ are independent component distributions and $\alpha_i$ are the individual probabilities assigned to the corresponding component distributions such that $\sum_{i=1}^{m} \pi_i = 1$.

It's assumed that all the arms in mixture model bandit environments follow the distribution defined above. What's flexible are the parameters of the component distributions and the categorical probabilities assigned to these components.

# 2. Known Variance

To start with, let's first assume that the variances of each arm are known to the learner and also that the component distributions are Gaussian. This means for each arm the reward obtained follows the distribution

$$f_X(x) = \sum_{i=1}^{m} \pi_i g_i(x), \quad \text{s.t. } g_i(x) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2\right\} \text{ for } i = 1 \to m,$$

where $\sigma_1, \ldots, \sigma_K$ are known to the learner throughout the plays.

## 2.1. The Upper Confidence Bound Algorithm

One very commonly used and well-studied algorithm for stationary stochastic bandits is the UCB algorithm, namely the upper confidence bound algorithm. Before adapting it to account for the mixture model assumption, this section aims to establish the basic assumptions of the UCB algorithm and its theoretical basis.

### 2.1.1. Assumptions and Theoretical Basis for UCB

One assumption that must be met in order to use UCB algorithm is sub-gaussianity.

**Definition 2.** *(Lattimore and Szepesvári, 2020) A random variable $X$ is $\sigma$-subgaussian if for all $\lambda \in \mathbb{R}$, it holds that $\mathbb{E}[\exp(\lambda X)] \leq \exp\left(\lambda^2\sigma^2/2\right)$.*

Gaussian variables are clearly sub-gaussian themselves by the definition and the moment generating function of the gaussian. Theorem 4 explains the origin of the term 'subgaussian'. The tails of a $\sigma$-subgaussian random variable decay approximately as fast as that of a Gaussian with zero mean and the same variance.

Note that for random variables that are not centred ($\mathbb{E}[X] \neq 0$), we abuse notation by saying that $X$ is $\sigma$-subgaussian if the noise $X - \mathbb{E}[X]$ is $\sigma$-subgaussian. A distribution is called $\sigma$-subgaussian if a random variable drawn from that distribution is $\sigma$-subgaussian. Subgaussianity is in fact a property to be possibly satisfied by both a random variable and the measure on the space on which it is defined (Lattimore and Szepesvári, 2020).

**Lemma 3.** *For any random variable $X$ and $\varepsilon > 0$, the following holds:*

(a) *(Markov):* $\mathbb{P}(|X| \geq \varepsilon) \leq \frac{\mathbb{E}[|X|]}{\varepsilon}$.

(b) *(Chebyshev):* $\mathbb{P}(|X - \mathbb{E}[X]| \geq \varepsilon) \leq \frac{\mathbb{V}[X]}{\varepsilon^2}$.

We need the above lemmas when proving the theorem 4, which provides a bound for the tail behaviour of sub-gaussian random variables.

**Theorem 4.** *(Lattimore and Szepesvári, 2020) If $X$ is $\sigma$-subgaussian, then for any $\varepsilon \geq 0$ ,*

$$\mathbb{P}(X \geq \varepsilon) \leq \exp\left(-\frac{\varepsilon^2}{2\sigma^2}\right)$$

*Proof.* We take a generic approach called the Cramér-Chernoff method. Let $\lambda > 0$ be some constant to be tuned later. Then

$$\begin{aligned}
\mathbb{P}(X \geq \varepsilon) &= \mathbb{P}(\exp(\lambda X) \geq \exp(\lambda\varepsilon)) \\
&\leq \mathbb{E}[\exp(\lambda X)]\exp(-\lambda\varepsilon) \quad \text{(Markov's inequality)} \\
&\leq \exp\left(\frac{\lambda^2\sigma^2}{2} - \lambda\varepsilon\right) \quad \text{(Def. of subgaussianity)}
\end{aligned}$$

Choosing $\lambda = \varepsilon/\sigma^2$ completes the proof. □

**Lemma 5.** *(Lattimore and Szepesvári, 2020) Suppose that $X$ is $\sigma$-subgaussian and $X_1$ and $X_2$ are independent and $\sigma_1$ and $\sigma_2$-subgaussian, respectively, then :*

(a) *$\mathbb{E}[X] = 0$ and $\mathbb{V}[X] \leq \sigma^2$.*

(b) *$cX$ is $|c|\sigma$-subgaussian for all $c \in \mathbb{R}$.*

(c) *$X_1 + X_2$ is $\sqrt{\sigma_1^2 + \sigma_2^2}$-subgaussian.*

**Corollary 6.** *(Lattimore and Szepesvári, 2020) Assume that $X_i - \mu$ are independent, $\sigma$-subgaussian random variables. Then for any $\varepsilon \geq 0$ ,*

$$\mathbb{P}(\hat{\mu} \geq \mu + \varepsilon) \leq \exp\left(-\frac{n\varepsilon^2}{2\sigma^2}\right) \quad \text{and} \quad \mathbb{P}(\hat{\mu} \leq \mu - \varepsilon) \leq \exp\left(-\frac{n\varepsilon^2}{2\sigma^2}\right)$$

*where $\hat{\mu} = \frac{1}{n}\sum_{t=1}^{n} X_t$.*

*Proof.* By Lemma 5, it holds that $\hat{\mu} - \mu = \sum_{i=1}^{n}(X_i - \mu)/n$ is $\sigma/\sqrt{n}$-subgaussian. Then apply Theorem 4. □

The above results is equivalent to the statement that under the assumptions of the result, for any $\delta \in [0, 1]$, with probability at least $1 - \delta$ (Lattimore and Szepesvári, 2020):

$$\mu \leq \hat{\mu} + \sqrt{\frac{2\sigma^2 \log(1/\delta)}{n}} \tag{2.1}$$

### 2.1.2. The Optimism Principle

The UCB algorithm is built upon *the principle of optimism in the face of uncertainty*, which states that a player should choose actions as if the environment is as nice as plausibly possible. The principle is useful not only for the finite-armed stochastic bandit problem (Lattimore and Szepesvári, 2020).

For bandits, the optimism principle translates to using the data (the rewards from each arm) observed so far to label each arm with a value, called the upper confidence bound that with overwhelming probability is an overestimate of the unknown mean. Note that the upper confidence bounds are updated after each play. The intuitive reason why this algorithm works is not hard to see. If the upper confidence bound assigned to the optimal arm is indeed an overestimate, then another arm can only be played if its upper confidence bound is greater than that of the optimal arm, which is itself larger than the mean of the optimal arm. And yet this cannot happen too frequently since data obtained by playing a suboptimal arm will provide more information about that sub-optimal arm. The upper confidence bound for this arm will eventually fall below that of the optimal arm.

In order to make the argument rigorous, it's necessary to define the upper confidence bound. Let $(X_t)_{t=1}^n$ be a sequence of independent 1-subgaussian random variables with mean $\mu$ and $\hat{\mu} = \frac{1}{n}\sum_{t=1}^n X_t$. By Eq. 2.1,

$$\mathbb{P}\left(\mu \geq \hat{\mu} + \sqrt{\frac{2\log(1/\delta)}{n}}\right) \leq \delta \quad \text{for all } \delta \in (0, 1). \tag{2.2}$$

When deciding its possible actions in play $t$, the learner has observed $T_i(t-1)$ samples from arm $i$ and received rewards from that arm with an empirical mean of $\hat{\mu}_i(t-1)$. Then a reasonable candidate 'as large as plausibly possible' for the unknown mean of the $i$ th arm can be defined as:

$$\mathrm{UCB}_i(t-1, \delta) = \begin{cases} \infty & \text{if } T_i(t-1) = 0 \\ \hat{\mu}_i(t-1) + \sqrt{\frac{2\log(1/\delta)}{T_i(t-1)}} & \text{otherwise} \end{cases} \tag{2.3}$$

To sum up, the UCB algorithm can therefore be presented as follows:

**Algorithm 1:** Upper Confidence Bound Algorithm (Lattimore and Szepesvári, 2020)

**Data:** $k$ and $\delta$
For $t = 1$, sample each arm once ;
**for** $t \in 2, \ldots, n$ **do**
    Choose action $A_t = \text{argmax}_i \, \text{UCB}_i(t-1)$;
    Observe reward $X_{A_t}$ and update upper confidence bounds;
**end**

## 2.2. Method 1 - without EM*

The first method for tackling this problem is to deliberately ignore the mixture-model assumption. This is based on the fact that mixture distributions may inherit some properties from their component distributions, for example if both component distributions are bounded then the mixture distribution will also be bounded. In this case, a mixture of subgaussian random variables is also subgaussian by the following derivation. This establishes that UCB (Upper Confidence Bound) algorithm can be also be applied here with assumptions properly satisfied.

For now, the assumption would be that the variances of $m$ component distributions are known. Suppose that an arm has a mixture model reward $X$ with $m$ components $X_1, \ldots, X_m$ with given variances $\sigma_1^2, \ldots, \sigma_m^2$ but unknown means $\mu_i$ and weights $\pi_i$. Then, we can derive an upper bound for the sub-gaussian parameter of $X$ with the definition of sub-gaussian as in Def. 2:

$$\begin{aligned}
\mathbb{E}\left[e^{\lambda X}\right] &= \int e^{\lambda x} \cdot f_x(x) dx \\
&= \int e^{\lambda x} \cdot \sum_{i=1}^{m} \pi_i f_{X_i}(x) dx \\
&= \sum_{i=1}^{m} \pi_i \int e^{\lambda x} f_{x_i}(x) dx \\
&= \sum_{i=1}^{m} \pi_i e^{\lambda^2 \sigma_i^2 / 2} \\
&\leqslant e^{\lambda^2 \max_i \{\sigma_i^2\}/2}
\end{aligned} \tag{2.4}$$

This means the reward here is at most $\max\{\sigma_1, \ldots, \sigma_m\}$-subgaussian by Def. 2 and also note that this value is likely an overestimate of the sub-gaussian parameter of the mixture model reward.

From Eq. 2.2, given $(X_t)_{t=1}^n$, a sequence of $\max\{\sigma_1, \ldots, \sigma_m\}$-subgaussian random variables, the inequality can be established as:

$$\mathbb{P}\left(\mu \geq \hat{\mu} + \max_i\{\sigma_i\}\sqrt{\frac{2\log(1/\delta)}{n}}\right) \leq \delta \quad \text{for all } \delta \in (0,1)$$

From Eq. 2.3, this can be rewritten in this form to account for the times that each arm $i$ has been sampled at time $t$, $T_i(t-1)$:

$$\text{UCB}_i(t-1,\delta) = \hat{\mu}_i(t-1) + \max_i\{\sigma_i\}\sqrt{\frac{2\log(1/\delta)}{T_i(t-1)}} \tag{2.5}$$

For asymptotically optimal UCB, a better regret bound can be achieved by replacing the confidence level term $\delta$ with a function of $t$ (Lattimore and Szepesvári, 2020).

$$\text{UCB}_i(t-1) = \hat{\mu}_i(t-1) + \max_i\{\sigma_i\}\sqrt{\frac{2\log f(t)}{T_i(t-1)}}, \text{ where } f(t) = 1 + t\log^2(t) \tag{2.6}$$

This derived bound can then be implemented in the aforementioned UCB algorithm, which hopefully will give us a low cumulative regret.

## 2.3. Method 2 - Expectation Maximization Algorithm*

The other method would be to consider the mixture model assumption and apply the EM (expectation-maximization algorithm) within a bandit algorithm to fit the mixture reward distribution of each arm.

### 2.3.1. Introduction to EM Algorithm

In statistics, an expectation-maximization (EM) algorithm is an iterative method to find (local) maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models, where the model depends on unobserved or so-called latent variables. The EM iterates between a expectation (E) step, which creates a function for the expectation of the log-likelihood evaluated using the current estimate for the parameters, and a maximization (M) step, which computes parameters maximizing the expected log-likelihood found on the E step. These parameter-estimates are then used to determine the distribution of the latent variables in the next E step. A formal description of the procedure firstly introduced by Dempster, Laird, and Rubin (1977) is given below:

Given the statistical model which generates a set $\mathbf{X}$ of observed data, a set of unobserved latent data or missing values $\mathbf{Z}$, and a vector of unknown parameters $\boldsymbol{\theta}$, along with a likelihoo $L(\boldsymbol{\theta}; \mathbf{X}, \mathbf{Z}) = p(\mathbf{X}, \mathbf{Z} \mid \boldsymbol{\theta})$, the maximum likelihood estimate (MLE) of the unknown parameters is determined by maximizing the marginal likelihood of the observed data:

$$L(\boldsymbol{\theta}; \mathbf{X}) = p(\mathbf{X} \mid \boldsymbol{\theta}) = \int p(\mathbf{X}, \mathbf{Z} \mid \boldsymbol{\theta}) d\mathbf{Z} = \int p(\mathbf{Z} \mid \mathbf{X}, \boldsymbol{\theta}) p(\mathbf{X} \mid \boldsymbol{\theta}) d\mathbf{Z}$$

However, this quantity is almost always intractable since $\mathbf{Z}$ is unobserved and the distribution of $\mathbf{Z}$ is unknown before attaining $\boldsymbol{\theta}$

The EM algorithm seeks to find the MLE of the marginal likelihood by iteratively applying these two steps (Dempster, Laird, and Rubin, 1977):

(a) Expectation step (E step): Define $Q\left(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}\right)$ as the expected value of the log likelihood function of $\boldsymbol{\theta}$, with respect to the current conditional distribution of $\mathbf{Z}$ given $\mathbf{X}$ and the current estimates of the parameters $\boldsymbol{\theta}^{(t)}$ :

$$Q\left(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}\right) = \mathrm{E}_{\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{(t)}}[\log L(\boldsymbol{\theta}; \mathbf{X}, \mathbf{Z})]$$

(b) Maximization step (M step): Find the parameters that maximize this quantity:

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} Q\left(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}\right)$$

### 2.3.2. EM for Gaussian Mixture Model

It's assumed in this project that the reward from each arm follows the Gaussian Mixture Model, abbreviated as GMM. In this section, the component variances of each arm are known to the player (but not the weights $\pi_i$ and variances $\sigma_i^2$).

There are many existing methods for estimating the parameters of a GMM, but EM based on the maximum likelihood estimation is perhaps the most appropriate one to be applied here.

For instance, consider the case where a dataset consists of many points that happen to be generated by two different distributions. The samples for each process follow a Gaussian distribution, but the data is mixed and thus forming a mixture. The distributions are overlapped and close enough that it is not apparent to see which distribution a given sample may originate from.

The stochasticity generating the data point from different component distributions is represented by a latent random variable. It influences the data but is not observable. As such, the EM algorithm is an appropriate approach to use to estimate the parameters of the distributions.

In the EM algorithm, the estimation-step try to find a value of the latent variable for each sample, and the maximization step would try to search for the optimal parameters of the probability distributions to try to best explain the data. The process is iterated until a set threshold has been met. The algorithm can be summarized as:

(a) E-Step. Estimate the expected value for each latent variable. This can be shown to be equivalent to calculating the probability that sample $i$ belongs to the cluster

$k$, which can be denoted as $w_{ik}$, which is calculated using Bayesian theorem as illustrated below. Here, $z_{ik}$ stands for the latent variable indicating that the sample $i$ comes from the cluster $k$, $\Theta$ the parameters of the model ($\mu$, $\Sigma$), $X$ the Gaussian mixture model random variable of interest.

$$\begin{aligned} w_{ik} &= p(z_{ik}=1|\underline{x}_i,\Theta) = \frac{p(X=\underline{x}_i|z_{ik}=1,\Theta)p(z_{ik}=1)}{p(X=x_i|\Theta)} \\ &= \frac{p_k(\underline{x}_i|z_k,\mu_k,\Sigma_k)\cdot\alpha_k}{\sum_{m=1}^{K} p_m(\underline{x}_i|z_m,\mu_m,\Sigma_m)\cdot\alpha_m} \end{aligned}$$

(b) M-Step. Optimize the parameters of the distribution using maximum likelihood. In the known variance case, only the means and weights $\pi_i$ are being iteratively updated (the first two terms). In a more common unknown variance case, update the parameters by:

$$\pi_k = \frac{\sum_{n=1}^{N} w_{nk}}{N}, \quad \mu_k^* = \frac{\sum_{n=1}^{N} w_{nk}\mathbf{x}_n}{\sum_{n=1}^{N} w_{nk}}, \quad \left(\Sigma_k^* = \frac{\sum_{n=1}^{N} w_{nk}\left(\mathbf{x}_n-\mu_k\right)\left(\mathbf{x}_n-\mu_k\right)^T}{\sum_{n=1}^{N} w_{nk}}\right)$$

### 2.3.3. Using EM in the UCB Algorithm*

One can apply EM within the UCB bandit algorithm to fit each arm. Suppose we have $m$ components of Gaussian. If we can use EM algorithm to estimate the weights $\hat{\pi}_i$, then an accurate estimate for variance can be established. Suppose the we have a mixture model reward $X$ with $m$ components $X_1,\ldots,X_m$ with weights $\pi_1,\ldots,\pi_m$ such that $\sum_{i=1}^{m}\pi_i = 1$. Suppose the components are all centered and variances are denoted as $\sigma_1^2,\ldots,\sigma_m^2$. Then,

$$\begin{aligned} \mathrm{E}[X] = \mu = \sum_{i=1}^{m}\pi_i\mu_i = 0, \\ \mathrm{E}\left[(X-\mu)^2\right] = \sigma^2 = \mathrm{E}\left[X^2\right] - \mu^2 = \sum_{i=1}^{m}\pi_i\mathrm{E}\left[X_i^2\right] = \sum_{i=1}^{m}\pi_i\sigma_i^2. \end{aligned} \tag{2.7}$$

From this, we know that the mixture model reward here is at least $\sqrt{\sum_{i=1}^{m}\pi_i\sigma_i^2}$-subgaussian. This is apparently a better estimate for the sub-gaussian parameter, compared to what we have derived before ignoring the mixture model assumption: $\max\{\sigma_1,\ldots,\sigma_m\}$-subgaussian.

Therefore, this can be used to give a better upper confidence bound with estimated weights $\hat{\pi}_i$ in a similar fashion as in Eq. 2.3

$$\mathrm{UCB}_i(t-1,\delta) = \begin{cases} \infty & \text{if } T_i(t-1)=0 \\ \hat{\mu}_i(t-1) + \sum_{i=1}^{m}\hat{\pi}_i\sigma_i^2\sqrt{\frac{2\log(1/\delta)}{T_i(t-1)}} & \text{otherwise} \end{cases} \tag{2.8}$$

Similarly, this can be adapted to the asymptotically optimal UCB as in Eq. 2.6:

$$\text{UCB}_i(t-1) = \hat{\mu}_i(t-1) + \sum_{i=1}^{m} \hat{\pi}_i \sigma_i^2 \sqrt{\frac{2\log f(t)}{T_i(t-1)}}, \text{ where } f(t) = 1 + t\log^2(t) \tag{2.9}$$

Then, the algorithm can be summarized in the following page. Note that as we have known variance in this case, the only thing that EM algorithm is improving the estimation of is the weights $\pi_i$. The parameters that are being updated in the EM algorithm are $\pi_i$ and $\mu_i$ for $i \in 1, \ldots, m$. However, the component means are not being used here as the estimates for the means of rewards because the simple sample mean $\frac{1}{N}\sum_{i=1}^{N} X_i$ is already an unbiased estimator for the mean. Therefore, there is no need to use a more sophisticated expression $\sum_{i=1}^{m} \pi_i \mu_i$ that also involves the output from EM algorithm as the estimate for the mean of the rewards.

**Algorithm 2:** UCB using EM (known variance)

**Input :** $x_, \ldots, x_N$, acc (accuracy), $n$ (the horizon), $k$ (the number of arms), $m$ (the number of mixture components per arm), $\sigma_{ij}^2$ (the variances) for $i = 1 \to k$ and $j = 1 \to m$

**for** $t = 1, \ldots, k$ **do** `// Initialization`

- Observe from arm $t$ for $m$ times and observe $X_{t,1}, \ldots, X_{t,m}$.
- Initialize $\bar{X}_{t,m}$ as $\frac{1}{m}\sum_{i=1}^{m} X_{t,i}$.
- Initialize $\pi_{t,j}$ randomly for $j = 1 \to m$.

**end**

**for** $t = km + 1, \ldots, n$ **do** `// Standard UCB-EM`

- Choose the action $\mathcal{A}_t$ that maximizes the UCB bound as in Eq. 2.9.
- Write $\mathcal{A}_t = i$ as the action, and observe $X_t$ from the arm $i$.
- Update $\hat{\pi}_{ij}$ for $j = 1 \to m$ using the EM algorithm given $X_{i,1}, \ldots, X_{i,T_i(t)}$ and $\sigma_{ij}^2$ for $j = 1 \to m$ (here $T_i(t)$ stands for the number of previous samples from the arm $i$).
- Update $\hat{\mu}_i(t)$ as $\frac{1}{T_i(t)}\sum_{j=1}^{T_i(t)} X_{i,j}$ (samples from the arm $i$ until time $t$).

**end**

Note that the initialization stage is necessary because we need at least some samples to start with in order to use the EM algorithm. If the number of samples to start with is too small, we have two problems. Firstly is that the program would not allow it (the packacge `sklearn.mixture.GaussianMixture` is used for the case when the variance is **unknown**, in the **known** variance case the EM is coded up by myself to account for the known variance feature). Secondly is that even when the algorithm converges, the $\pi_i$ is likely to be stuck in the local minimum 0 or 1, which is undesirable.

## 2.4. Empirical Results for Known Variance Case*

In this section, the empirical results about the performances of the two algorithms aforementioned are given: the one ignoring the mixture model structure, and the one using EM. Note that in all the plots afterwards, psuedo-regrets are used as a means of comparing the performances of algorithms (policies) as described in Eq. 1.1.

$$\bar{R}_n = n\mu^* - \sum_{t=1}^{n} \mu_{A_t} \qquad \text{(pseudo-regret)}$$

For all the plots afterwards, the parameters information is written in the titles:

- **k**: the number of arms.
- **n**: the horizon (number of plays).
- **m**: the number of component distributions (in the initial case of mixture distribution, $m = 2$)
- **r**: the number of repeated experiments under the same bandit environment.
- $\boldsymbol{a < \mu < b}$: this indicates that the means of all the component distributions of this bandit environment are sampled from the uniform distribution over $[a, b]$.
- $\boldsymbol{a < \sigma < b}$: this indicates that the standard deviations of all the component distributions of this bandit environment are sampled from the uniform distribution over $[a, b]$.

For each bandit environment, the experiment is repeated for 30 times because the regret is generated stochastically and so is the reward from each arm. A 95% confidence interval is generated while the middle line stands for the mean regret over 30 experiments.

The main observation here is that UCB-EM performs uniformly better and beats UCB in almost all the bandit environments. Each page is afterwards is denoted to a fixed number of components. For example, for figure 2.1 to figure 2.6, the reward is a mixture of 2 gaussians, i.e., $f_X(x) = \alpha f_1(x) + (1 - \alpha) f_2(x)$.

In the 6 plots drawn each page, the left 3 ones is devoted to a more challenging bandit environment with means of gaussians initialized from the range $[0, 1]$, while for the right 3 plots the means are generated from the range $[0, 5]$. The reason why the these particular bandit environments are deemed easy or difficult is not hard to see: imagine a 2-arm bandit problem such that the first arm keeps giving rewards between 0 and 1 while the second arm only gives very high reward randomly in the range 100 and 101. This is arguably a very easy bandit problem if we assume that the rewards are random uniform variates. In other words, the bandit problem is deemed more difficult to minimize regret if the mean rewards of arms are more clustered together and close to each other.

The UCB-EM algorithm approximately takes 10 times more time to run compared to UCB alone. Even though the computational efforts are definitely bigger with the implementation of EM algorithm, the efforts pay off in the regret and have somehow improved the performance of the UCB policy in almost all bandit environments in which experiments have done.

One other observation is that in a less challenging bandit environments (the right 3 plots on each page), the gap between the regrets of these algorithms are visibly smaller. That's because in an easier bandit environment, as explained before, both of the algorithms can learn the optimal arm in relatively less time, thus overshadowing the benefits brought by a more accurate estimate of the variance term by EM algorithm. Thus, one can choose to use either UCB or UCB-EM in a more careless way if the bandit problem is deemed 'easy'.

There can be some way of detecting whether a bandit problem is easy or difficult. If a UCB-based method is implemented, the relative ratios among the number of plays for different machine can be used as a criterion. If we want to use something flashy, **the Gini coefficient** is a possible choice here as originally a measure for the statistical dispersion. A player can firstly do a number of plays, and then calculate the Gini coefficient to see whether all the rewards are more or less coming from a single arm or are actually alternating among different arms, indicating that the algorithm finds it difficult to locate the optimal arm. A subsequent decision on which algorithm to use can then be made. As the rewards and actions are recorded as history, there would be no big problems stemming from changing an algorithm to another during the plays as long as the player is trying to minimize the cumulative regret.

Another observable trend is that when the number of components $m$ increases, so is the difference in performances of the UCB and UCB-EM algorithms under a more challenging bandit environment. The EM algorithm really has done more efforts in estimating a more accurate variance when the number of components increases. However, as $m$ increases, the difference in performance becomes less distinguishable when the bandit environment is easy.

The following figures compare the UCB and UCB-EM algorithm with $m = 2$ (2 components).

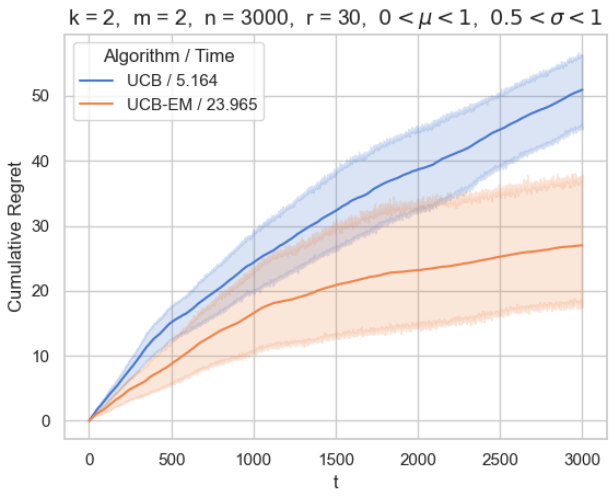


**Figure 2.1**

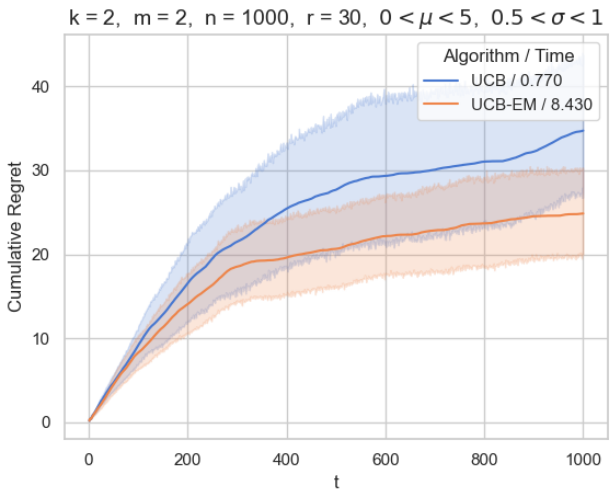


**Figure 2.2**

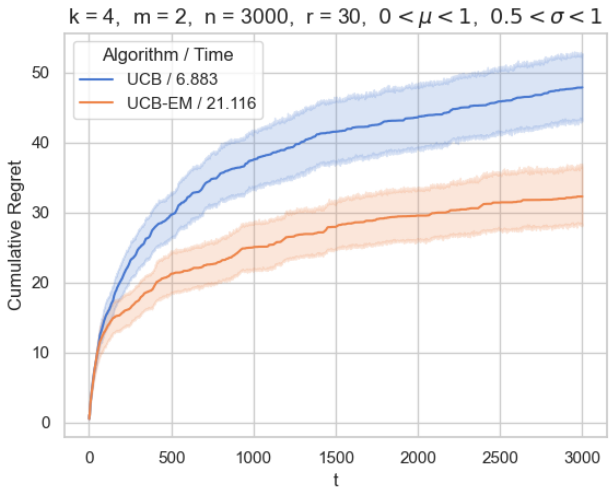


**Figure 2.3**

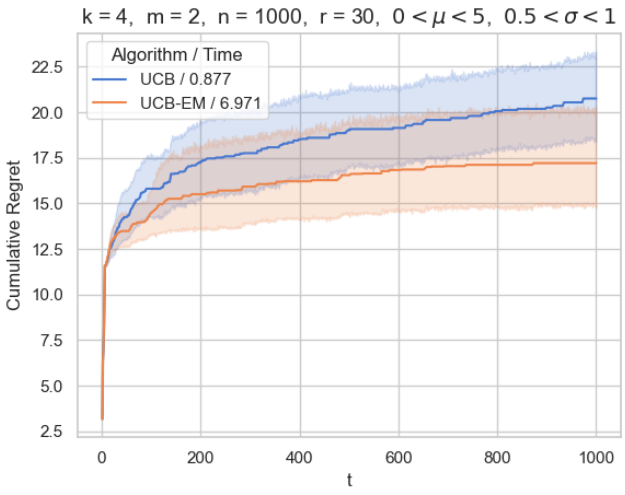


**Figure 2.4**

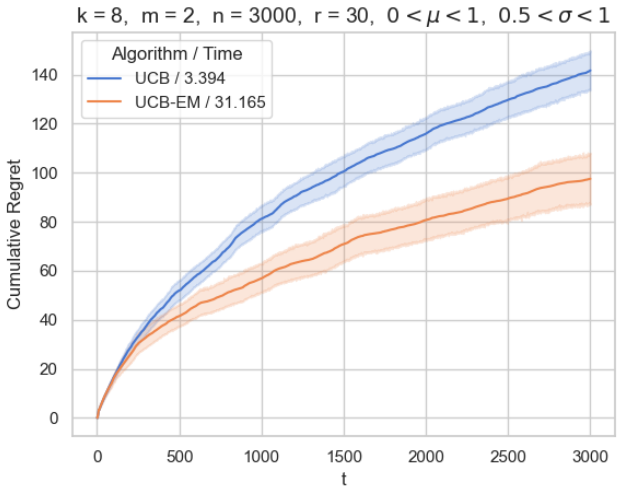


**Figure 2.5**

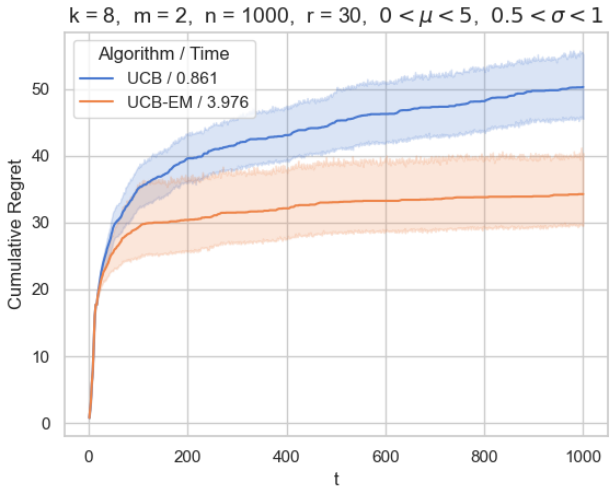


**Figure 2.6**

The following figures compare the UCB and UCB-EM algorithm with $m = 3$ (3 components).

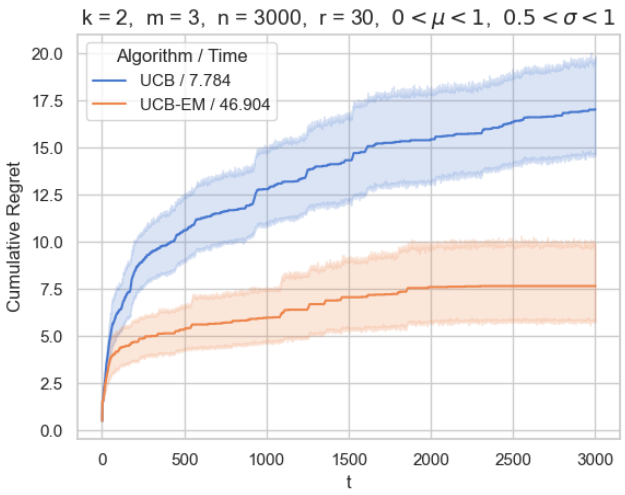


**Figure 2.7**

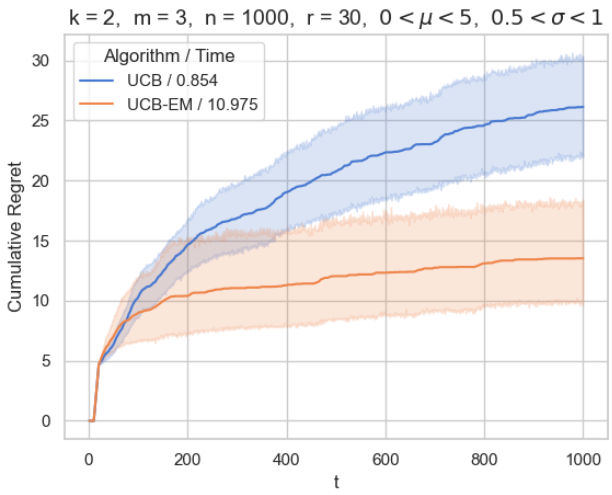


**Figure 2.8**

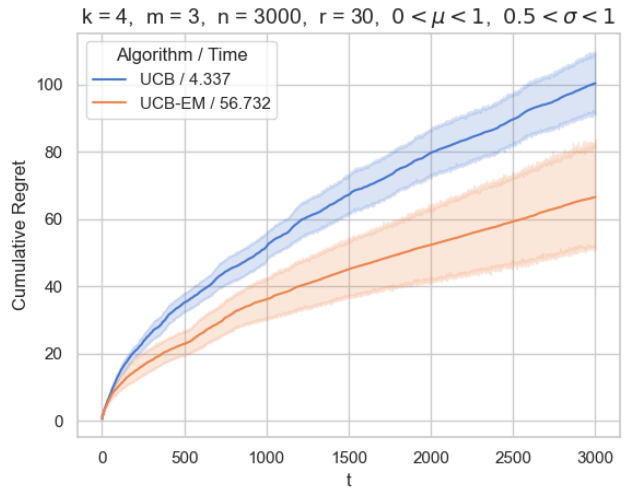


**Figure 2.9**

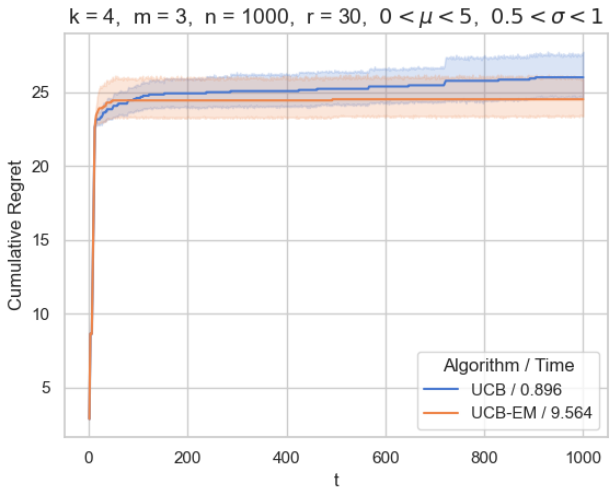


**Figure 2.10**

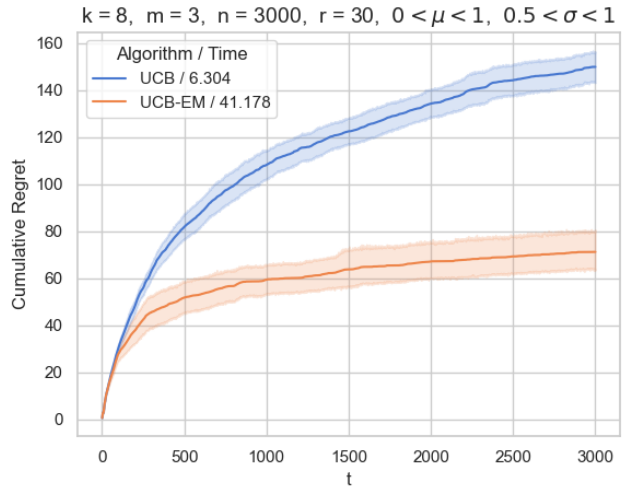


**Figure 2.11**

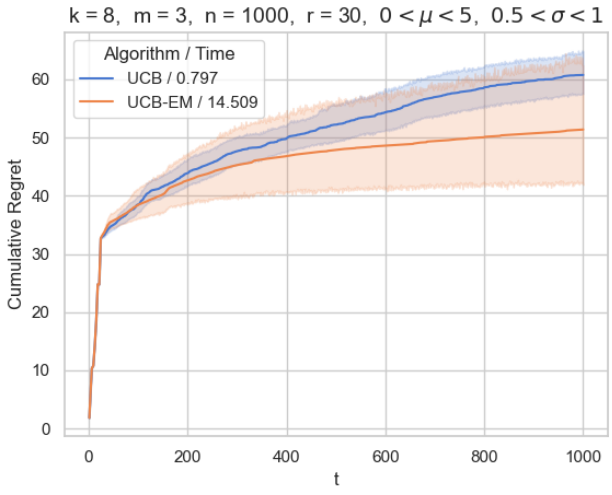


**Figure 2.12**

The following figures compare the UCB and UCB-EM algorithm with $m = 4$ (4 components).

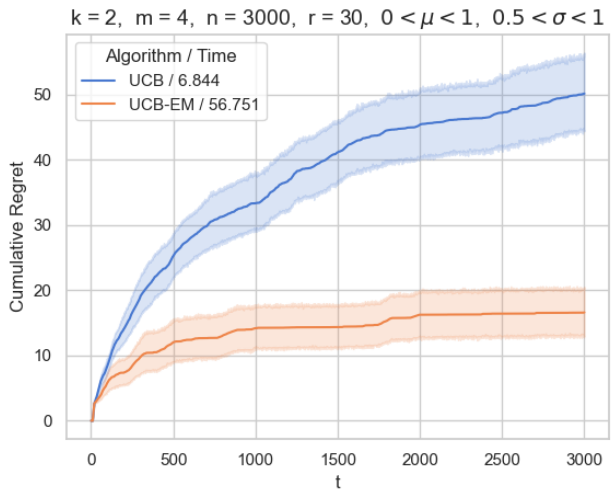


**Figure 2.13**

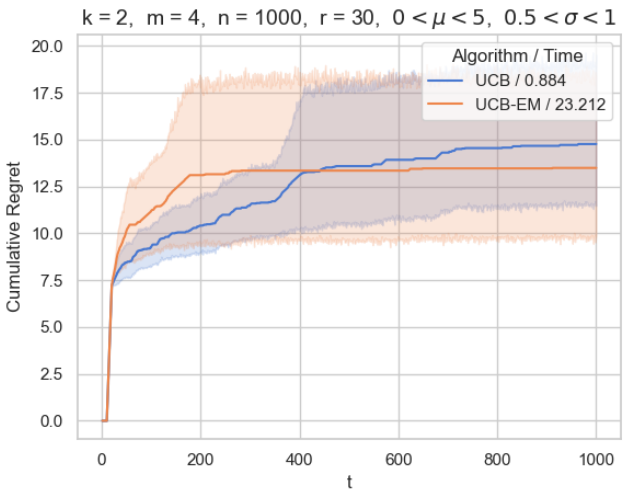


**Figure 2.14**

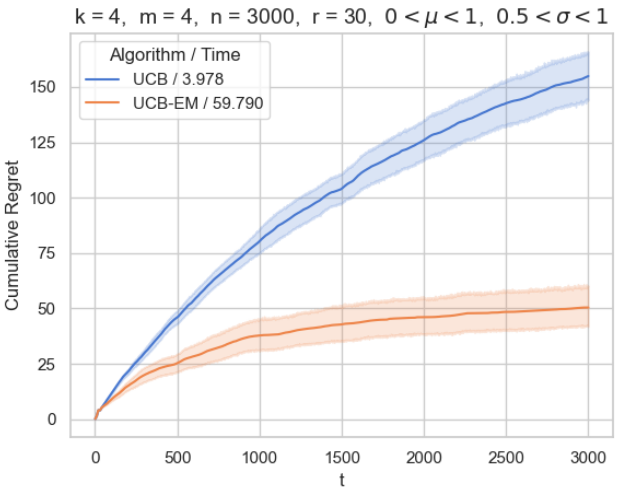


**Figure 2.15**

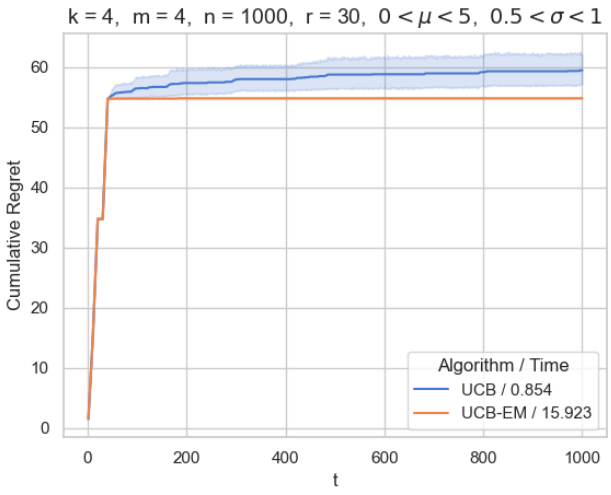


**Figure 2.16**

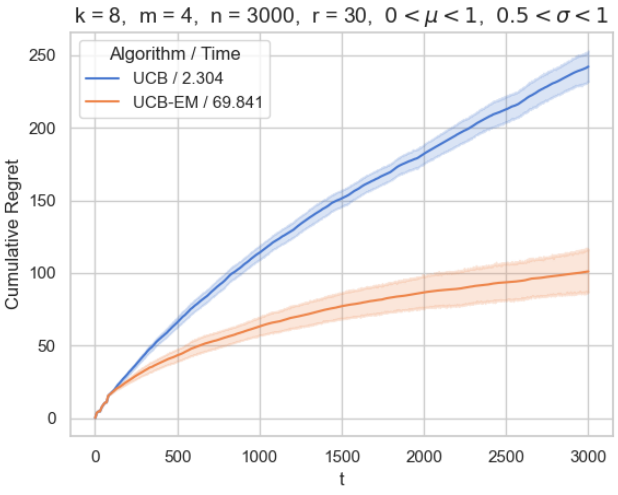


**Figure 2.17**

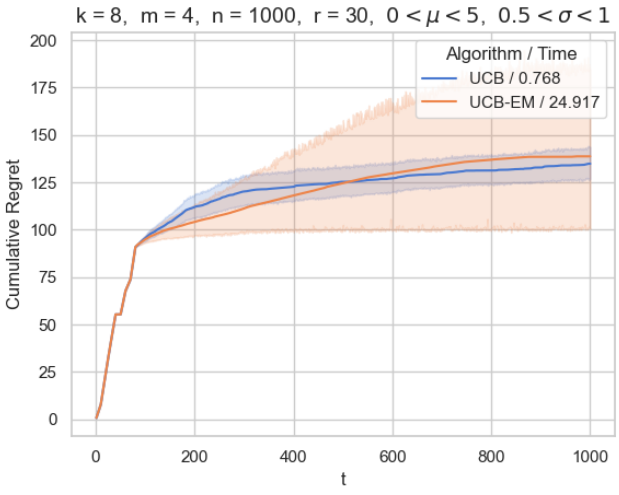


**Figure 2.18**

# 3. Unknown Variance Case

In this section, it's assumed that no prior information about the parameters of the arms is given to the player. Aside from the mixture model and Gaussian assumptions formulated previously, the means, variances, and weights of the component distributions are all unknown. For each arm, the reward $X$ obtained follows a mixture model distribution

$$f_X(x) = \sum_{i=1}^{m} \pi_i g_i(x), \quad \text{s.t. } g_i(x) = \frac{1}{\sigma_i\sqrt{2\pi}} \exp\left\{-\frac{1}{2}\left(\frac{x-\mu_i}{\sigma_i}\right)^2\right\} \text{ for } i = 1 \to m.$$

## 3.1. UCB-V (Empirical Bernstein UCB)

UCB-V algorithm is introduced in the paper by Audibert, Munos, and Szepesvári (2007). It is also commonly referred to as the empirical Bernstein UCB because it has used the Bennett's and Bernstein's inequalities in deriving the upper confidence bound and the regret bound. It is particularly useful here because the algorithm assumes no information is given about the variance of the rewards. Empirical estimates for variances and means are used here. EM algorithm can also be adapted to provide a better estimate for sample variance.

### 3.1.1. Assumptions for Using UCB-V

The assumptions of the algorithm are: Let $K > 2, \nu_1, \ldots, \nu_K$ distributions over reals with support $[0, b]$. For $1 \leq k \leq K$, let $\{X_{k,t}\} \sim \nu_k$ be an i.i.d. sequence of random variables specifying the rewards for arm $k$. Assume that the rewards of different arms are independent of each other, i.e., for any $k, k', 1 \leq k < k' \leq K, t \in \mathbb{N}^+$, the collection of random variables, $(X_{k,1}, \ldots, X_{k,t})$ and $(X_{k',1}, \ldots, X_{k',t})$, are independent of each other (Audibert, Munos, and Szepesvári, 2007).

### 3.1.2. The UCB-V Algorithm

For any $k \in \{1, \ldots, K\}$ and $t \in \mathbb{N}$, let $\bar{X}_{k,t}$ and $V_{k,t}$ be the empirical estimates of the mean payoff and variance of arm $k$ :

$$\bar{X}_{k,t} \triangleq \frac{1}{t}\sum_{i=1}^{t} X_{k,i} \quad \text{and} \quad V_{k,t} \triangleq \frac{1}{t}\sum_{i=1}^{t}\left(X_{k,i} - \bar{X}_{k,t}\right)^2 \tag{3.1}$$

Let $c \geq 0$. Let $\mathcal{E} = (\mathcal{E}_{s,t})_{s\geq 0, t\geq 0}$ be non-negative real numbers such that for any $s \geq 0$, the function $t \mapsto \mathcal{E}_{s,t}$ is non-decreasing. $\mathcal{E}$ (a function of $(s,t)$) is called the exploration function. For any arm $k$ and any non-negative integers $s, t$, introduce

$$B_{k,s,t} \triangleq \bar{X}_{k,s} + \sqrt{\frac{2V_{k,s}\mathcal{E}_{s,t}}{s}} + c\frac{3b\mathcal{E}_{s,t}}{s} \tag{3.2}$$

with the convention $1/0 = +\infty$.

**UCB-V algorithm**: At any time $t$, play an arm maximizing $B_{k,T_k(t-1),t}$ (Audibert, Munos, and Szepesvári, 2007).

### 3.1.3. Natural Choices for the Hyperparameters in the UCB-V Algorithm*

As we can see from the formula in Eq. 3.2, it involves unspecified exploration function $\epsilon$ and $c$. In the paper by Audibert, Munos, and Szepesvári (2007), it's been pointed out that these are hyperparameters to be tuned under different environment. However, the choice can't be chosen without any constraint.

It's been established and proved by Audibert, Munos, and Szepesvári (2007) that in order to achieve sub-linear and logarithmic regret in particular, natural values for the constants appearing in the bound are:

$$B_{k,s,t} \triangleq \bar{X}_{k,s} + \sqrt{\frac{2V_{k,s}\log t}{s}} + \frac{b\log t}{2s} \tag{3.3}$$

This choice corresponds to the critical exploration function $\mathcal{E}_t = \log t$ and to $c = 1/6$, which is also the minimal associated value of $c$ to ensure that the UCB-V algorithm doesn't suffer a polynomial loss. The proofs that lead to this result are beyond the scope of this project.

### 3.1.4. The UCB-V Algorithm for Mixture Model Bandits*

Here, UCB-V algorithm can be directly applied to the mixture model bandit problem with a minor breach of the assumptions. The empirical variances can be directly estimated by the sample variance as defined in Eq. 3.1.

We require a value of $b$ in the Eq. 3.1, an upper bound for the range of the rewards of **all** arms, which doesn't allow us to use the UCB-V algorithm here strictly speaking . However, with the sub-gaussian property of the mixture gaussian random variables as discussed before, we can assume that the tail behaviors of the mixture of gaussians

allows us to believe most of the samples from this mixture distribution would stay in an approximated interval $[\hat{b}_1, \hat{b}_2]$. Then, we can use this estimated $b_2$ to replace the $b$ in Eq. 3.3. In practice, I choose to estimate the $b$ by the maximum value of the rewards we get from playing any arms, i.e., $\hat{b} = \max_{k,s}\{X_{k,s} \mid \forall k, \forall s\}$. As a result, we can expect $\hat{b}$ to be reasonably close to what we need here eventually for a horizon that is sufficiently large.

Note that there is **no** theoretical guarantee or regret bound obtained for this revised algorithm, but empirically this works well as shown in the cumulative regret plots given in the following sections.

**Algorithm 3:** UCB-V for Mixture Model Bandits without EM

**Input :** $x_, \ldots, x_N$, $n$ (the horizon), $k$ (the number of arms), $m$ (the number of mixture components per arm)

**for** $t = 1, \ldots, n$ **do**

Choose the action $\mathcal{A}_t$ that maximizes the UCB bound as defined in Eq. 3.3.

Denote $i = \mathcal{A}_t$ as the action, and observe $X_{i,T_i(t-1)}$.

Update $\bar{X}_{i,T_i(t-1)}$ and $\bar{V}_{i,T_i(t-1)}$ as defined in Eq. 3.1

Update $\hat{b} = \max_{k,s}\{X_{k,s} \mid \forall k, \forall s\}$

**end**

### 3.1.5. The UCB-V Algorithm for Mixture Model Bandits with EM*

The next algorithm to be considered here is an 'advancement' for what we have done in the previous section. We want to consider the mixture model assumption here. Therefore, EM algorithm can be used to provide an estimate for the empirical variance term $\bar{V}_{k,s}$ with assumption about the mixture model reward for each arm.

Suppose the means of each components are $\mu_1, \ldots, \mu_m$ and variances are $\sigma_1^2, \ldots, \sigma_m^2$, then we know:

$$
\begin{aligned}
\mathrm{E}[X] = \mu &= \sum_{i=1}^{m} \pi_i \mu_i \\
\mathrm{E}\left[(X - \hat{\mu})^2\right] &= \sigma^2 = \mathrm{Var}[X] \\
&= \mathrm{E}[X^2] - \mu^2 \\
&= \sum_{i=1}^{m} \pi_i \cdot \mathrm{E}[X_i^2] - \mu^2 \\
&= \sum_{i=1}^{m} \pi_i \left(\sigma_i^2 + \mu_i^2\right) - \mu^2
\end{aligned}
\tag{3.4}
$$

The empirical variance can therefore be estimated as:

$$
\bar{V}_{k,t} \triangleq \sum_{i=1}^{m} \hat{\pi}_i \left(\hat{\sigma}_i^2 + \hat{\mu}_i^2\right) - \hat{\mu}^2, \tag{3.5}
$$

where $\hat{\pi}_1$ are weights, $\hat{\sigma}_i^2$ and $\hat{\mu}_i$ are component variances and means, which can be estimated from the EM algorithm with samples $X_{k,1}, \ldots, X_{k,t}$.

By a similar way of implementing EM within the UCB-V algorithm, we call this algorithm UCB-V-EM algorithm, which is summarized as below:

**Algorithm 4:** UCB-V for Mixture Model Bandits with EM

**Input :** $x_, \ldots, x_N$, $n$ (the horizon), $k$ (the number of arms), $m$ (the number of mixture components per arm)

**for** $t = 1, \ldots, k$ **do** `// Initialization`

- Observe from arm $t$ for $m$ times and observe $X_{t,1}, \ldots, X_{t,m}$.
- Initialize $\bar{X}_{t,m}$ as $\frac{1}{m}\sum_{i=1}^{m} X_{t,i}$ and $\bar{V}_{t,m}$ as 1.
- Initialize $\pi_{t,j}$ randomly for $j = 1 \to m$.

**end**

**for** $t = km, \ldots, n$ **do** `// Standard UCB-V-EM Steps`

- Choose the action $\mathcal{A}_t$ that maximizes the UCB bound as defined in Eq. 3.3.
- Denote $\mathcal{A}_t = i$ as the action, and observe $X_{i,T_i(t-1)}$.
- Update $\bar{X}_{i,T_i(t-1)}$ as defined in Eq. 3.1.
- Update $\bar{V}_{i,T_i(t-1)}$ as defined in Eq. 3.5 through EM algorithm.
- Update $\hat{b} = \max_{k,s}\{X_{k,s} \mid \forall k, \forall s\}$.

**end**

## 3.2. The UCB1-Normal Algorithm for Mixture Model Bandits*

Another policy considered here is the UCB1-NORMAL algorithm introduced by Auer, Cesa-Bianchi, and Fischer (2002). It's designed for the multi-armed bandit problem un-

der the assumptions that each machine has a reward that follows a normal distribution with unknown means and variances.

The algorithm is especially concerned with the finite-armed bandit problem with Gaussian rewards. It's been shown by Auer, Cesa-Bianchi, and Fischer (2002) that UCB1-NORMAL has logarithmic regret. The proof has used bounds on the tails of the $\chi^2$ and the Student distribution that has only been verified numerically.

As our assumption is that the reward follows a mixture of Gaussian, it's therefore compelling to use UCB1-NORMAL algorithm here because we can reasonably expect that the tail behaviours of mixture of Gaussian are similar to those of a single normal distribution. Therefore, the UCB1-NORMAL algorithm is directly used here to assess its performance in the mixture model bandit problem.

Again, the interest here is whether considering the mixture model assumption of the reward would lead to an improvement of the performance of the different algorithms, and also about the extra computational cost induced by the EM algorithm. Therefore, the sample variance term in the Eq. 3.7 in the UCB1-NORMAL algorithm, $\frac{q_j - n_j \bar{x}_j^2}{n_j - 1}$, can be replaced by the one estimated using EM algorithm as we have derived before in Eq. 3.5:

$$\bar{V}_{k,t} \triangleq \sum_{i=1}^{m} \hat{\pi}_i \left( \hat{\sigma}_i^2 + \hat{\mu}_i^2 \right) - \hat{\mu}^2, \tag{3.6}$$

**Algorithm 5:** UCB1-NORMAl (Auer, Cesa-Bianchi, and Fischer, 2002)

**for** $n = 1, 2, \ldots$ **do**

If there is an arm that has been played less than $\lceil 8 \log n \rceil$ times, play this arm.

Otherwise, play arm $j$ that maximizes

$$\bar{x}_j + \sqrt{16 \cdot \frac{q_j - n_j \bar{x}_j^2}{n_j - 1} \cdot \frac{\ln(n-1)}{n_j}} \tag{3.7}$$

where $\bar{x}_j$ is the average reward obtained from machine $j$, $q_j$ is the sum of squared rewards obtained from machine $j$, and $n_j$ is the number of times machine $j$ has been played so far.

Update $\bar{x}_j$ and $q_j$ with the obtained reward $x_j$

**end**

**Algorithm 6:** Altered UCB1-NORMAL with EM

**for** $n = 1, 2, \ldots$ **do**

If there is an arm that has been played less than $\lceil 8 \log n \rceil$ times, play this arm.

Otherwise, play arm $j$ that maximizes

$$\bar{x}_j + \sqrt{16 \cdot \bar{\mathbf{V}}_{\mathbf{j},\mathbf{n_j}} \cdot \frac{\ln(n-1)}{n_j}}$$

where $\bar{x}_j$ is the average reward obtained from machine $j$, $\hat{\sigma}_j^2$ is the estimated sample variance from machine $j$, $n_j$ is the number of times machine $j$ has been played so far, and $\bar{V}_{j,n_j}$ is the empirical variance estimated from using the $n_j$ samples from arm $j$ as written in Eq. 3.5.

Update $\bar{x}_j$ and $\hat{\sigma}_j^2$ with the obtained reward $x_j$ using EM algorithm.

**end**

## 3.3. Empirical Results and Regret Comparison for Unknown Variance

This section presents the empirical results by comparing the cumulative regrets of these policies that assume unknown variance: UCB-V, UCB-V-EM, UCB1-NORMAL, and UCB1-NORMAL-EM.

The regret used here is same as previous sections, which is the pseudo-regret as discussed in Eq. 1.1: $\bar{R}_n = n\mu^* - \sum_{t=1}^{n} \mu_{A_t}$. The notations in the titles of the figures have the same meanings as those in the previous figures (see page 19 for details) .

From observations of the plots, the main conclusion here is to **never** use UCB1-NORMAL or UCB1-NORMAL-EM under mixture model assumptions. In all the plots, UCB1-NORMAL, no matter whether implemented with EM or not, can't achieve the regret as low as the UCB-V algorithm. This is probably caused by the first step of the UCB1-NORMAL algorithm, which selects an arm to play whenever it's been played less than $\lceil 8 \log n \rceil$ times. Even though the algorithm achieves logarithmic regrets under Gaussian setting, this step might result in large regret because it chooses an arm not based on its upper confidence bound. The ruggedness of the cumulative regret of the UCB1-NORMAL algorithm is also caused by this.

However, the EM algorithm did somehow play a role in improving the regret of the UCB1-NORMAL algorithm, even though the UCB1-NORMAL algorithm is not fit to be used in the mixture model setting. This can be observed most clearly in the figure 3.4, 3.9, 3.11, 3.13.

Unfortunately, EM didn't do a great job in reducing the regret of the UCB-V algorithm. The mean and the confidence interval of the cumulative regrets don't seem to be affected by the introduction of the EM. The expectation is that with the mixture model assumption taking into account, a better estimate for the variance can be attained. However, this result is also expected because we know that the sample variance is already itself an unbiased estimator for the overall variance of an arm. With the mixture model assumptions, only one piece of new information is introduced which is the number of components $m$. This is arguably quite limited if we assume that the variances are unknown.

The time taken by the UCB-V-EM algorithm to run is on average 10 times more than that of the single UCB-V algorithm. So, **the plain UCB-V is the optimal one to use** so far among the four aforementioned algorithms for mixture model bandits because it has the best regret and takes the smallest computational cost.

UCB-V, UCB-V-EM, UCB1-NORMAL, UCB1-NORMAL-EM, **2 arms**

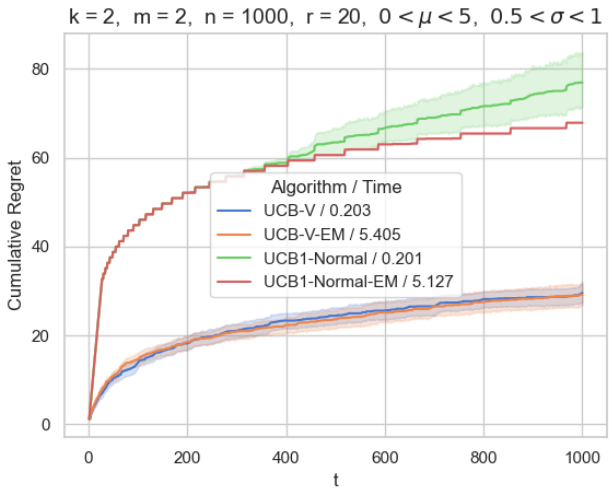


**Figure 3.1**

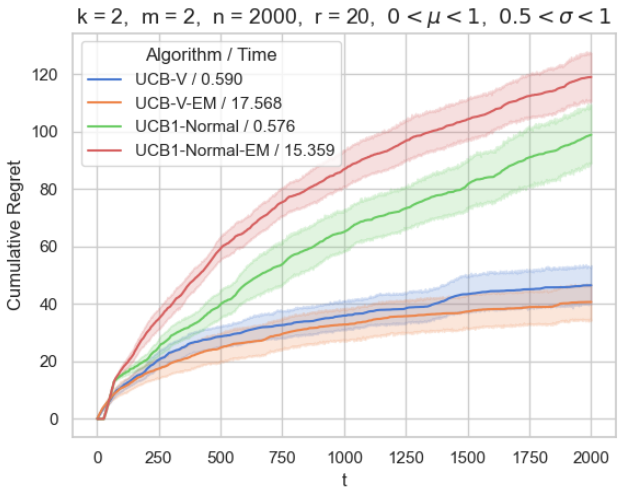


**Figure 3.2**

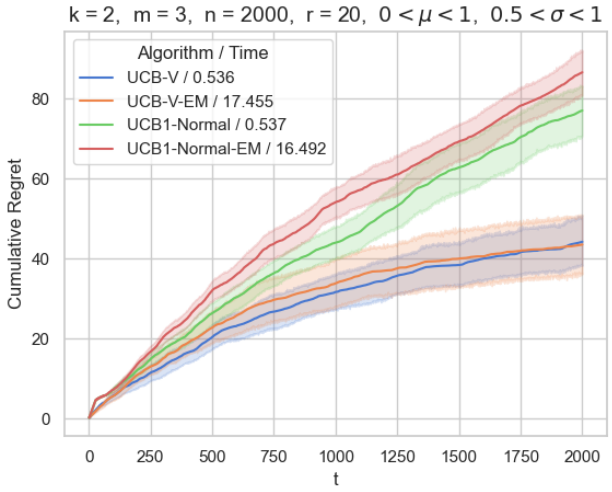


**Figure 3.3**

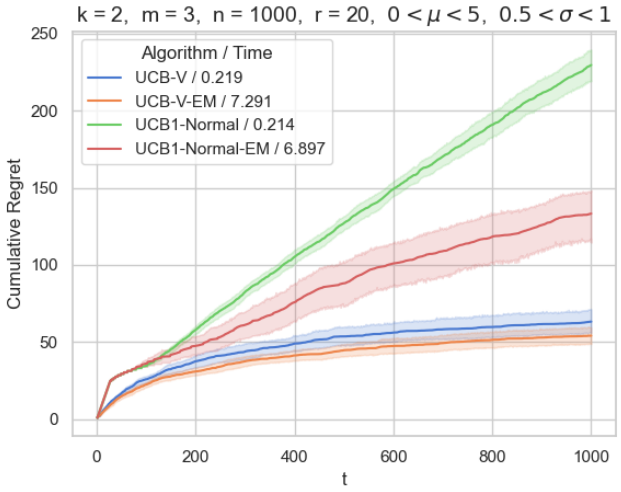


**Figure 3.4**

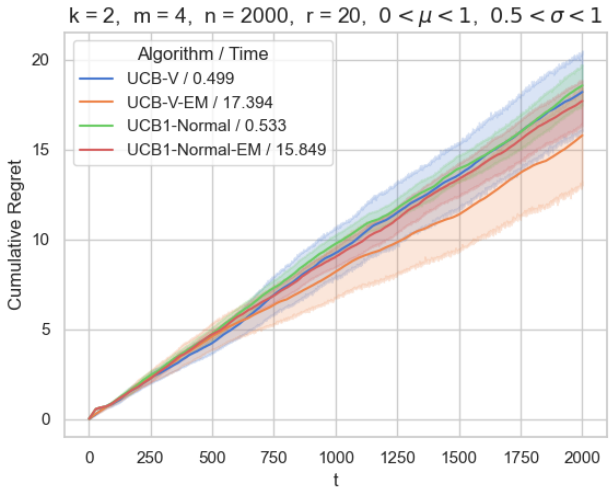


**Figure 3.5**

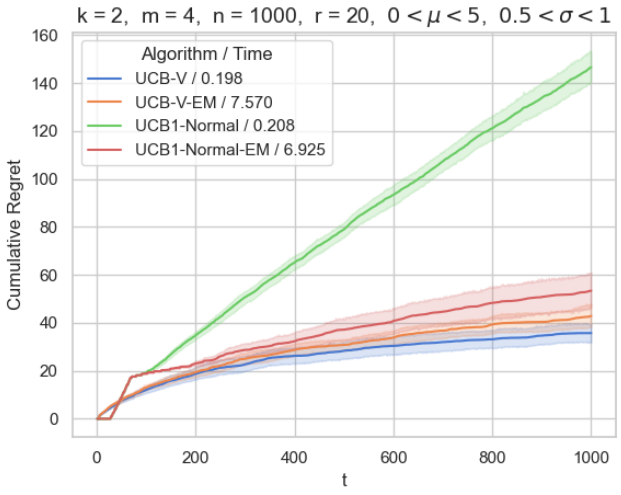


**Figure 3.6**

UCB-V, UCB-V-EM, UCB1-NORMAL, UCB1-NORMAL-EM, **4 arms**

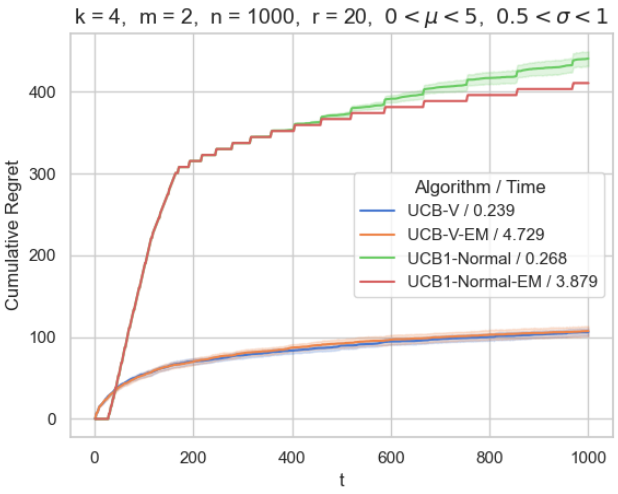


**Figure 3.7**

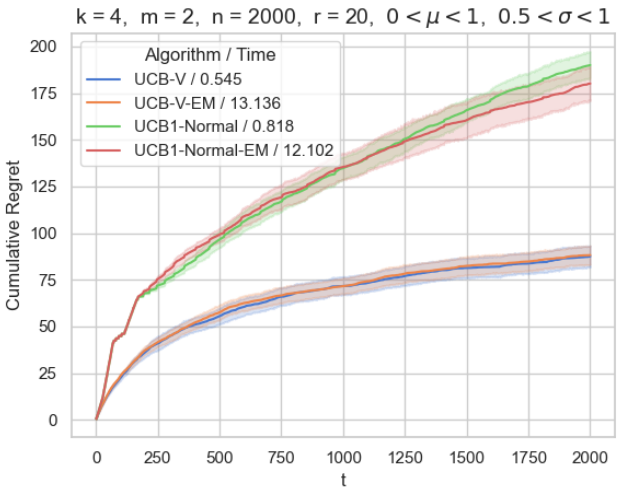


**Figure 3.8**

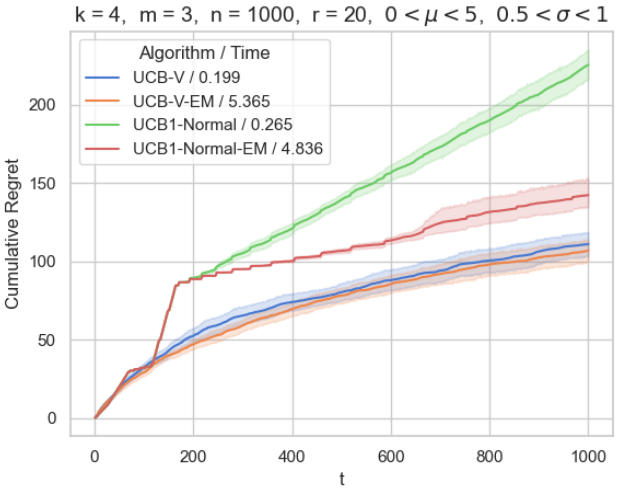


**Figure 3.9**

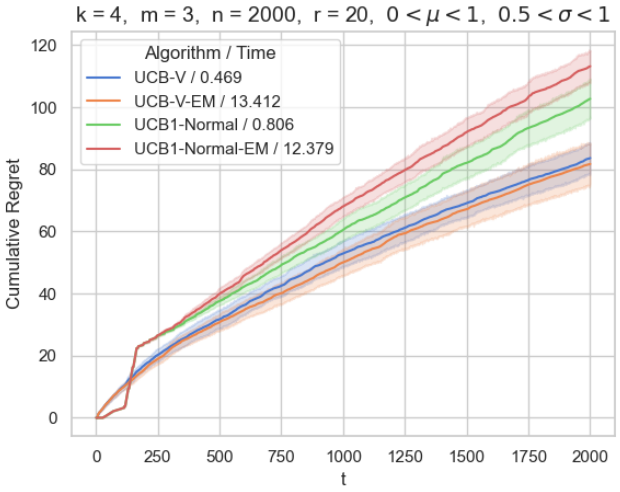


**Figure 3.10**

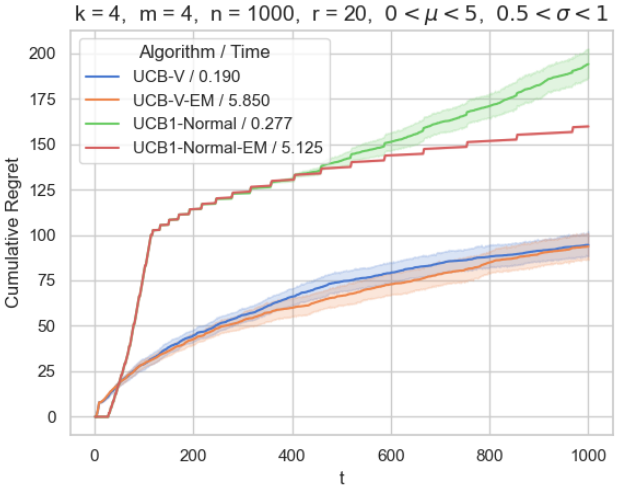


**Figure 3.11**

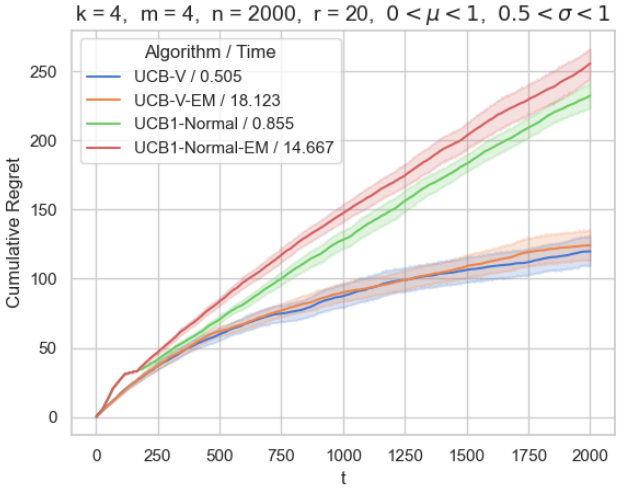


**Figure 3.12**

UCB-V, UCB-V-EM, UCB1-NORMAL, UCB1-NORMAL-EM, **8 arms**

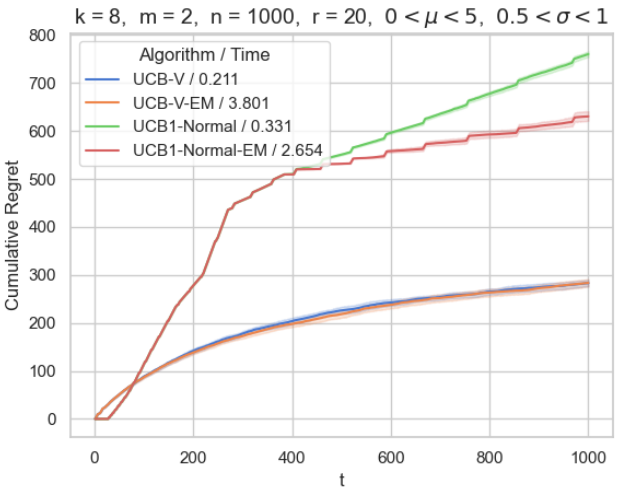


**Figure 3.13**

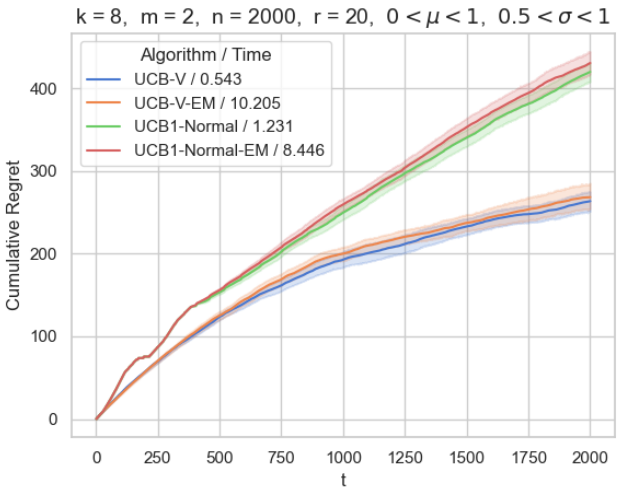


**Figure 3.14**

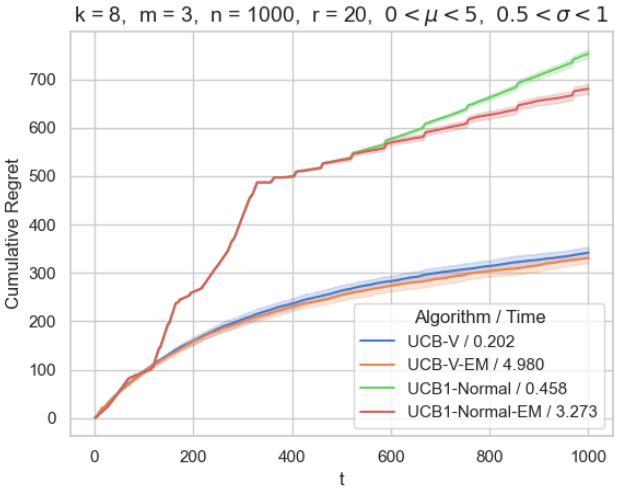


**Figure 3.15**

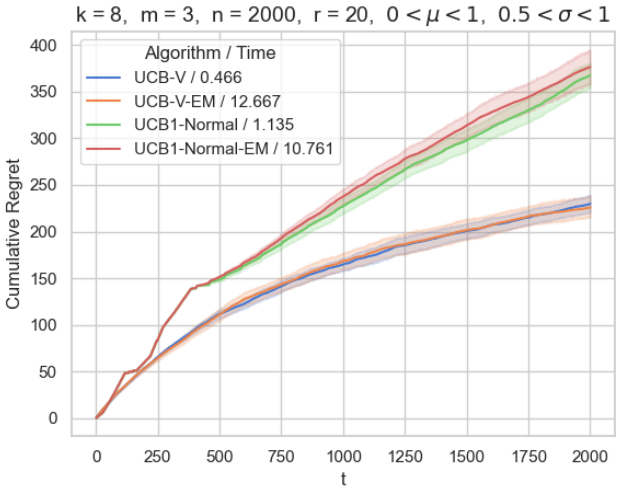


**Figure 3.16**

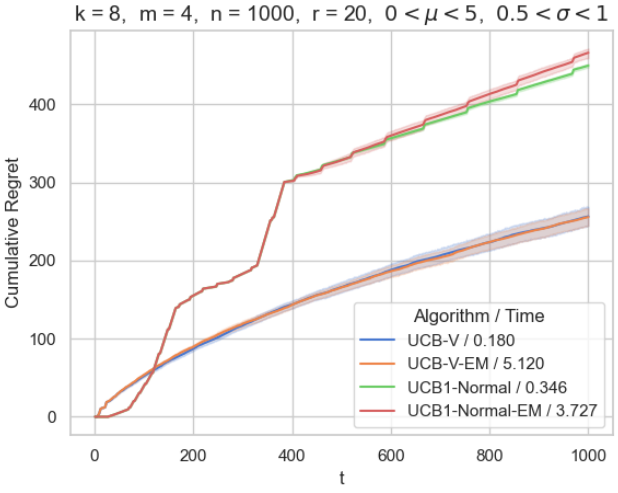


**Figure 3.17**

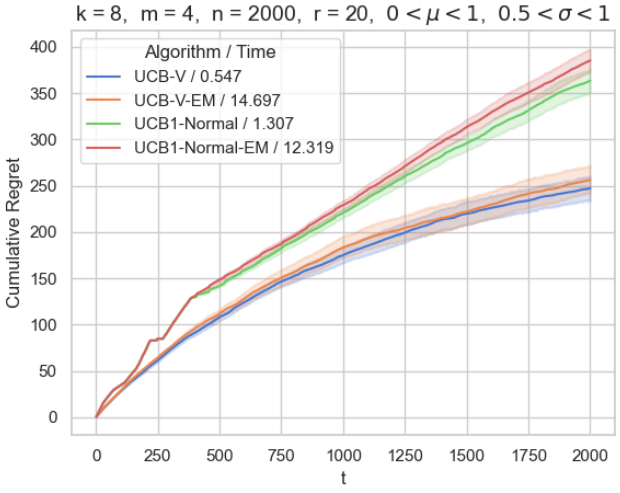


**Figure 3.18**

# 4. Thompson Sampling

In this section, it's assumed that the bandit environment is in a Bayesian setting. More specifically, the Bayesian bandit environment requires that the parameters of each arm are sampled from prior distributions. A more detailed definition is given in the following section.

The goal is to try to explore the performance of Thompson sampling used in the mixture model bandits and compare it with the UCB-V algorithm.

Note that it doesn't make sense here to 'ignore' the mixture model assumptions here because in Bayesian bandits a prior is placed on the original bandit environment which contains information about the component means, variances, and weights. It would be impractical to ignore this and place a prior on overall means and variances, etc., because that would be a violation against the assumption about the mixture model reward distribution.

In this section, it's also assumed that the component variances, means, weights are unknown and that the reward follows a mixture of independent Gaussian distributions.

## 4.1. The Bayesian Bandit Environment

The Bayesian bandit model is different in its own way that at the start of the bandit game, an environment is randomly sampled from the prior. Similarly, the sampled environment is not accessible to the learner. In this project, it's assumed we have only a finite number of arms.

**Definition 7** (Bayesian bandit environment, Lattimore and Szepesvári (2020))**.** *A* $k$*-armed Bayesian bandit environment is a tuple* $(\mathcal{E}, \mathcal{G}, Q, P)$*, where* $(\mathcal{E}, \mathcal{G})$ *is a measurable space and* $Q$ *is a probability measure on* $(\mathcal{E}, \mathcal{G})$ *called the prior. The last element* $P = (P_{\nu i} : \nu \in \mathcal{E}, i \in [k])$ *is a probability kernel from* $\mathcal{E} \times [k]$ *to* $(\mathbb{R}, \mathfrak{B}(\mathbb{R}))$*, where* $P_{\nu i}$ *is the reward distribution associated with the* $i$ *th arm in bandit* $\nu$*. A Bayesian bandit environment and policy* $\pi = (\pi_t)_{t=1}^n$ *interact to produce a collection of random variables,* $\nu \in \mathcal{E}, (A_t)_{t=1}^n$ *and* $(X_t)_{t=1}^n$ *with* $A_t \in [k]$ *and* $X_t \in \mathbb{R}$ *that satisfy:*

(a) $\mathbb{P}(\nu \in \cdot) = Q(\cdot)$

(b) the conditional distribution of action $A_t$ given $\nu, A_1, X_1, \ldots, A_{t-1}, X_{t-1}$ is $\pi_t(\cdot \mid A_1, X_1, \ldots, A_{t-1}, X_{t-1})$ almost surely; and

(c) the conditional distribution of the reward $X_t$ given $\nu, A_1, X_1, \ldots, A_t$ is $P_{\nu A_t}$ almost surely.

*Example* 8. (Lattimore and Szepesvári, 2020) A $k$-armed Bayesian Bernoulli bandit environment could be defined by letting $\mathcal{E} = [0,1]^k, \mathcal{G} = \mathfrak{B}(\mathcal{E})$ and $P_{\nu i} = \mathcal{B}(\nu_i)$. A natural prior in this case would be a product of Beta$(\alpha, \beta)$ distributions (Lattimore and Szepesvári, 2020):

$$Q(A) = \int_A \prod_{i=1}^{k} q_i(x_i)\, dx$$

where $q_i(x) = x^{\alpha-1}(1-x)^{\beta-1}\Gamma(\alpha+\beta)/(\Gamma(\alpha)\Gamma(\beta))$.

## 4.2. Posterior Distributions in Bayesian Bandits

Let $(\mathcal{E}, \mathcal{G}, Q, P)$ be a $k$-armed Bayesian bandit environment. Assuming that $(\mathcal{E}, \mathcal{G})$ is a Borel space, The following theorem guarantees the existence of the posterior: a probability kernel $Q(\cdot \mid \cdot)$ from the space of histories to $(\mathcal{E}, \mathcal{G})$ with the form

$$Q(A \mid a_1, x_1, \ldots, a_t, x_t) \tag{4.1}$$

which can also be written as $\mathbb{E}[\mathbb{I}_A(\nu) \mid A_1, X_1, \ldots, A_t, X_t]$ (Lattimore and Szepesvári, 2020).

**Theorem 9.** *(Lattimore and Szepesvári, 2020) If $(\Theta, \mathcal{G})$ is a Borel space, then there exists a probability kernel $Q : \mathcal{X} \times \mathcal{G} \to [0,1]$ such that $Q(A \mid X) = \mathbb{P}(\theta \in A \mid X)$ simultaneously for all $A \in \mathcal{G}$ outside some $P$-null set. Furthermore, for any two probability kernels $Q, Q'$ satisfying this condition, $Q(\cdot \mid x) = Q'(\cdot \mid x)$ for all $x$ in some set of $\mathbb{P}_X$-probability one.*

*Proof.* The proof of the above result is beyond the scope of this project. □

*Example* 10. (Lattimore and Szepesvári, 2020) The posterior for the Bayesian bandit in Example 8 in terms of its density with respect to the Lebesgue measure is

$$q(\theta \mid \underbrace{a_1, x_1, \ldots, a_t, x_t}_{h_t}) \propto \prod_{i=1}^{k} \theta_i^{\alpha + s_i(h_t) - 1} (1-\theta_i)^{\beta + t_i(h_t) - s_i(h_t) - 1}$$

where $s_i(h_t) = \sum_{u=1}^{t} x_u \mathbb{I}\{a_u = i\}$ and $t_i(h_t) = \sum_{u=1}^{t} \mathbb{I}\{a_u = i\}$. This means the posterior is also the product of Beta distributions, each updated according to the observations from the relevant arm.

## 4.3. The Bayesian Regret

We have defined before that the regret of policy $\pi$ in $k$-armed bandit environment $\nu$ over $n$ rounds is

$$R_n(\pi, \nu) = n\mu^* - \mathbb{E}\left[\sum_{t=1}^{n} X_t\right] \tag{4.2}$$

where $\mu^* = \max_{i\in[k]} \mu_i$ and $\mu_i$ is the mean of $P_{\nu i}$.

**Definition 11** (Bayesian Regret, Lattimore and Szepesvári (2020))**.** *Given a $k$-armed Bayesian bandit environment $(\mathcal{E}, \mathcal{G}, Q, P)$ and a policy $\pi$, the Bayesian regret is*

$$\mathrm{BR}_n(\pi, Q) = \int_{\mathcal{E}} R_n(\pi, \nu) dQ(\nu) \tag{4.3}$$

The Bayesian optimal regret is $\mathrm{BR}_n^*(Q) = \inf_\pi \mathrm{BR}_n(\pi, Q)$, and the optimal policy is

$$\pi^* = \mathrm{argmin}_\pi \, \mathrm{BR}_n(\pi, Q) \tag{4.4}$$

## 4.4. Introduction to Thompson Sampling

Thompson sampling, named after William R. Thompson, is an algorithm for choosing actions that addresses the exploration-exploitation dilemma in the multi-armed bandit problem. Intuitively, it states that the player should select the action that optimizes the expected reward with a randomly drawn belief based on previous actions and rewards history. The heuristic was firstly introduced by Thompson (1933).
Formally, let $(\mathcal{E}, \mathfrak{B}(\mathcal{E}), Q, P)$ be a $k$-armed Bayesian bandit environment as defined in the previous sections. The player then chooses actions $(A_t)_{t=1}^n$ and obtains rewards $(X_t)_{t=1}^n$, and the posterior given $t$ observations is a probability kernel $Q(\cdot \mid \cdot)\,\mathrm{from}([k] \times \mathbb{R})^t$ to $(\mathcal{E}, \mathfrak{B}(\mathcal{E}))$. Denote the mean of the $i$ th arm in bandit $\nu \in \mathcal{E}$ by $\mu_i(\nu) = \int_{\mathbb{R}} x dP_{\nu i}(x)$. In round $t$, Thompson sampling algorithm samples a bandit environment $\nu_t$ from the posterior of $Q$ given $A_1, X_1, \ldots, A_{t-1}, X_{t-1}$ and then chooses the arm with the largest mean.

**Algorithm 7:** Thompson Sampling (Lattimore and Szepesvári, 2020)

**Input :** Bayesian bandit environment $(\mathcal{E}, \mathfrak{B}(\mathcal{E}), Q, P)$
**for** $n = 1, 2, \ldots, n$ **do**
  Sample $\nu_t \sim Q\left(\cdot \mid A_1, X_1, \ldots, A_{t-1}, X_{t-1}\right)$
  Choose $A_t = \mathrm{argmax}_{i\in[k]} \, \mu_i\left(\nu_t\right)$
**end**

The validity and soundness of the Thompson sampling algorithm can also be reasoned as the existence of a bound for the Bayesian regret, similar to the UCB-based algorithms.

**Theorem 12** (Bayesian Regret of Thompson Sampling, Lattimore and Szepesvári (2020))**.** *Let $(\mathcal{E}, \mathfrak{B}(\mathcal{E}), Q, P)$ be a $k$-armed Bayesian bandit environment such that for all $\nu \in \mathcal{E}$*

*and $i \in [k]$, the distribution $P_{\nu i}$ is 1-subgaussian (after centring) with mean in $[0, 1]$ Then the policy $\pi$ of Thompson sampling satisfies*

$$\mathrm{BR}_n(\pi, Q) \leq C\sqrt{kn \log(n)}$$

*where $C > 0$ is a universal constant.*

*Proof.* The proof of the above result is beyond the scope of this project. □

## 4.5. Thompson Sampling for Mixture Model Bandits*

From the algorithm of Thompson sampling, it's obvious that the main task and challenge here is the calculation of posterior distribution $Q(\cdot \mid A_1, X_1, \ldots, A_{t-1}, X_{t-1})$. In the non-Bayesian setting discussed in the previous sections, the EM (Expectation and Maximization) algorithm is used to work out the parameters of a mixture distribution. Here, under a Bayesian setting, the problem is more complex because the posterior distribution is in most cases intractable with hidden states, in this case the labels of samples indicating which components it comes from.

The method to work out the posterior of a mixture of Gaussian is the *Variational Bayes* (VB) method, also known as variational inference methods. One python function, namely, `sklearn.mixture.Bayesian.GaussianMixture` has implemented the variational Bayesian method for graphical models introduced and discussed in the paper by Attias et al. (1999), and the VB inference for Dirichlet process mixtures discussed in the paper by Blei and Jordan (2006) for working out the posterior of a mixture of Gaussian, which is exactly what's needed here.

Even though the details about the derivation of this method and the proof of the validity and efficiency of the VB methods are beyond the focus and the scope of this project, a general idea about how it works is summarized in the following section.

### 4.5.1. Introduction to Variational Bayesian Methods

This section is an introduction to the basis of the mathematical derivation of variational Bayesian methods which will be used to estimate the posterior of the mixture of Gaussian in the Thompson sampling algorithm. One of the earliest literature that formulates the variational Bayes problems and introduces the VB algorithm and its generalizations can be found in the book by Beal (2003).

In variational Bayes methods, we seek to approximate the posterior distribution over a set of unobserved variables $\mathbf{Z} = \{Z_1 \ldots Z_n\}$ given data $\mathbf{X}$ by a variational distribution, $Q(\mathbf{Z})$:

$$P(\mathbf{Z} \mid \mathbf{X}) \approx Q(\mathbf{Z})$$

Therefore, the task is transformed to minimize the dissimilarity between these two distributions. It's common to use the Kullback-Leibler divergence (KL-divergence) of $P$ from $Q$ as the measure for dissimilarity.

$$D_{\mathrm{KL}}(Q\|P) \triangleq \sum_{\mathbf{Z}} Q(\mathbf{Z}) \log \frac{Q(\mathbf{Z})}{P(\mathbf{Z} \mid \mathbf{X})}.$$

By Bayes theorem, we know that

$$P(\mathbf{Z} \mid \mathbf{X}) = \frac{P(\mathbf{X} \mid \mathbf{Z})P(\mathbf{Z})}{P(\mathbf{X})} = \frac{P(\mathbf{X} \mid \mathbf{Z})P(\mathbf{Z})}{\int_{\mathbf{Z}} P\left(\mathbf{X}, \mathbf{Z}'\right) d\mathbf{Z}'}.$$

The difficulty in this problem stems from the fact the integral over $\mathbf{Z}$ to calculate the marginal distribution $P(\mathbf{X})$ in the denominator is often intractable (both computationally and analytically). Therefore, we seek an approximation, using $Q(\mathbf{Z}) \approx P(\mathbf{Z} \mid \mathbf{X})$.

Given that $P(\mathbf{Z} \mid \mathbf{X}) = \frac{P(\mathbf{X},\mathbf{Z})}{P(\mathbf{X})}$, the KL-divergence above can also be written as

$$\begin{aligned} D_{\mathrm{KL}}(Q\|P) &= \sum_{\mathbf{Z}} Q(\mathbf{Z}) \left[\log \frac{Q(\mathbf{Z})}{P(\mathbf{Z}, \mathbf{X})} + \log P(\mathbf{X})\right] \\ &= \sum_{\mathbf{Z}} Q(\mathbf{Z})[\log Q(\mathbf{Z}) - \log P(\mathbf{Z}, \mathbf{X})] + \sum_{\mathbf{Z}} Q(\mathbf{Z})[\log P(\mathbf{X})] \\ &= \sum_{\mathbf{Z}} Q(\mathbf{Z})[\log Q(\mathbf{Z}) - \log P(\mathbf{Z}, \mathbf{X})] + \log P(\mathbf{X}) \\ &= \mathbb{E}_{\mathbf{Q}}[\log Q(\mathbf{Z}) - \log P(\mathbf{Z}, \mathbf{X})] + \log P(\mathbf{X}), \end{aligned} \tag{4.5}$$

which can be rearranged to become

$$\log P(\mathbf{X}) = D_{\mathrm{KL}}(Q\|P) - \mathbb{E}_{\mathbf{Q}}[\log Q(\mathbf{Z}) - \log P(\mathbf{Z}, \mathbf{X})] = D_{\mathrm{KL}}(Q\|P) + \mathcal{L}(Q).$$

As $\log P(\mathbf{X})$ is constant with respect to $Q$, to minimize the $D_{\mathrm{KL}}(Q\|P)$, it's equivalent to maximize $\mathcal{L}(Q)$. By appropriate choice of $Q$, $\mathcal{L}(Q)$ would be tractable enough to maximize easily. The variational distribution $Q(\mathbf{Z})$ is usually assumed to factorize over some partition of the latent variables, i.e. for some partition of the latent variables $\mathbf{Z}$ into $\mathbf{Z}_1 \dots \mathbf{Z}_M$ (Attias et al., 1999; Beal, 2003):

$$Q(\mathbf{Z}) = \prod_{i=1}^{M} q_i \left(\mathbf{Z}_i \mid \mathbf{X}\right).$$

### 4.5.2. Python Implementation of the Thompson Sampling*

Information is given in this section about the python function `sklearn.mixture.BayesianGaussianMixture` which is implemented in the Thompson sampling algorithm.

The basic assumption here is that for each arm we have a mixture of $m$ independent gaussians as the reward. Then, the prior for means of component distributions

are themselves Gaussian $\mu_i \sim \mathcal{N}(\hat{\mu_i}, \bar{\sigma}^2)$. Here $\hat{\mu}$ can be worked out from k-means algorithm as the mean for the prior (this is done automatically in the program). Here $\bar{\sigma}$ is the prior knowledge we have about the variance of the mean distribution, which is denoted `mean_precision_prior` as the inverse of the variance and as a parameter in the `BayesianGaussianMixture` function.

The prior for covariance is the Wishart distribution as the default. As we are only dealing with one dimensional reward, Wishart distribution is equivalent to a Gamma distribution as the prior. A specific prior with customized parameters can be set with the option `covariance_prior`. Here, it's omitted so that it can be initialized automatically as the empirical covariance of the given samples.

`weight_concentration_prior_type` specifies the type of the prior for the mixture probabilities. Mixture models can be either finite or infinite. If it's infinite, a Dirichlet process is set as the prior. In this case, as only $m$ components are considered. Therefore, what's needed is only a finite version of the Dirichlet process, the Dirichlet distribution. Dirichlet distribution is the conjugate prior for the categorical distribution characteristic of the weights. Here, the prior of $\pi_i$ is set as the default $\text{Dir}(1/m, \ldots, 1/m)$ because it's assumed there are m components and the prior doesn't have a preference for one component over any of the other components.

After generating an instance from the `BayesianGaussianMixture` and fitting it with data, 3 attributes are used here to compute the posterior means, which are `means_`, `mean_precision_`, and `dirichlet_concentration`.

As a result, $\pi_i$ is sampled from the posterior Dirichlet distribution with concentration parameters `dirichlet_concentration`. $\mu_i$ is sampled from the posterior of the means, which is a gaussian distribution with parameters `means_` and `mean_precision_`. Subsequently, the posterior mean of a given bandit reward with $m$ components is equal to $\sum_{i=1}^{m} \pi_i \mu_i$.

Note that the posterior covariances of the components are not needed because the calculation of the mean of the Bayesian bandits doesn't involve the covariances.

*(More details about the usage of* ***`sklearn.mixture.BayesianGaussianMixture`*** *can be found at this link.)*

## 4.6. Empirical Results for Thompson Sampling*

This section contains the empirical results comparing Thompson sampling with UCB-V-EM. Only algorithms that assume the unknown variances and use EM are compared with Thompson sampling because it's desirable to keep the assumptions of algorithms as consistent as possible (even though not exactly the same). UCB-V-EM use EM, which is

a non-Bayesian version of the variational Bayes method used in the Thompson Sampling.

UCB1-NORMAL algorithms are not taken into consideration because they perform uniformly worse than the UCB-V under the mixture model assumptions.

The regret used here is same as previous sections, which is the pseudo-regret as discussed in Eq. 1.1: $\bar{R}_n = n\mu^* - \sum_{t=1}^{n} \mu_{A_t}$. The notations in the titles of the figures have the same meanings as those in the previous figures (see page 19 for more details) .

The experiments have done for a large variety of bandit environments. Experiments are done for 2, 4, and 6 arms cases. The number of components ranges from 2 to 4. For a more challenging bandit problem, the means of each component are sampled from a uniform distribution with support $[0, 1]$, compare to $[0, 5]$ for an easier counterpart. The possible standard deviations are all sampled uniformaly from $[0.5, 1]$ for the all the plots below.

The main result is that Thompson sampling overall has **much better** performance than the UCB-V-EM. The improvement in the performance is most evident in less challenging bandit environments (the left 3 plots on each page). In more difficult bandit problems, the reduction in cumulative regret is less significant. In addition, it seems like Thompson sampling has slightly larger confidence intervals in the cumulative regret plots for some cases (more noticeably in 2-armed cases).

Thompson sampling in general increases the running time more than by 20-fold, which means it requires significantly **more computational efforts** than UCB-V.

**2 arms case**, regret comparison for Thompson Sampling and UCB-V-EM

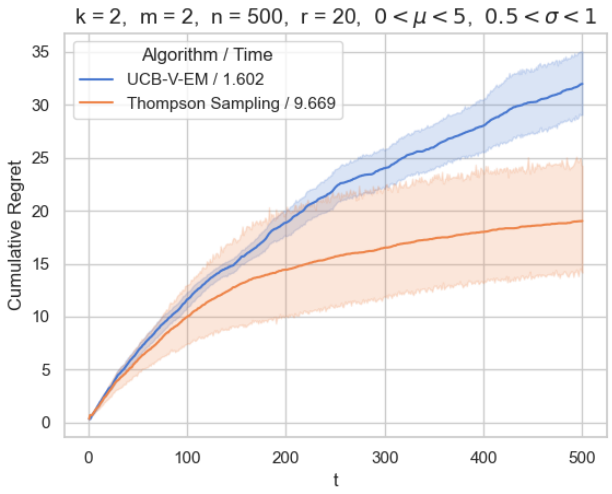


**Figure 4.1**

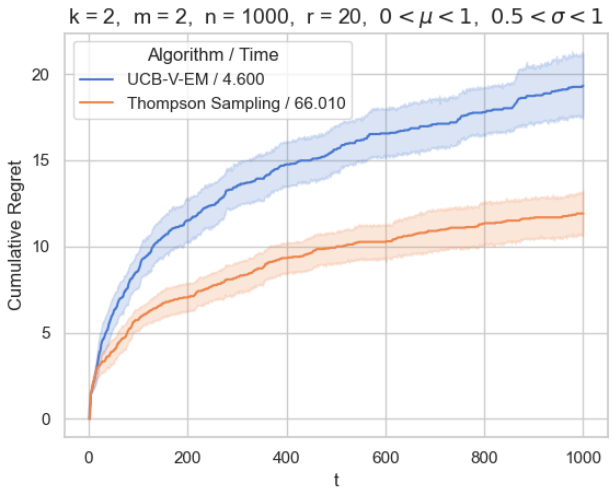


**Figure 4.2**

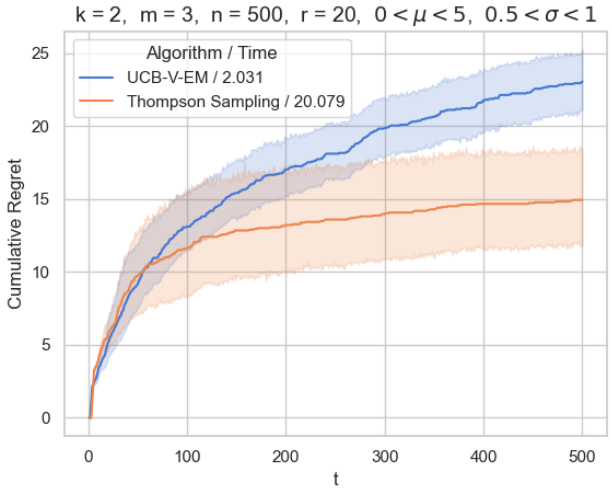


**Figure 4.3**

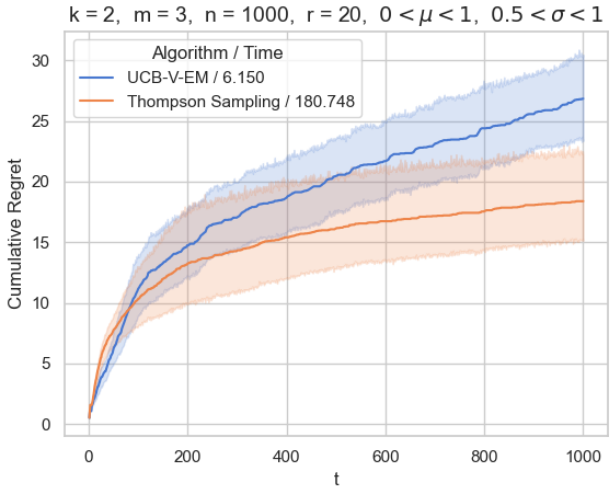


**Figure 4.4**

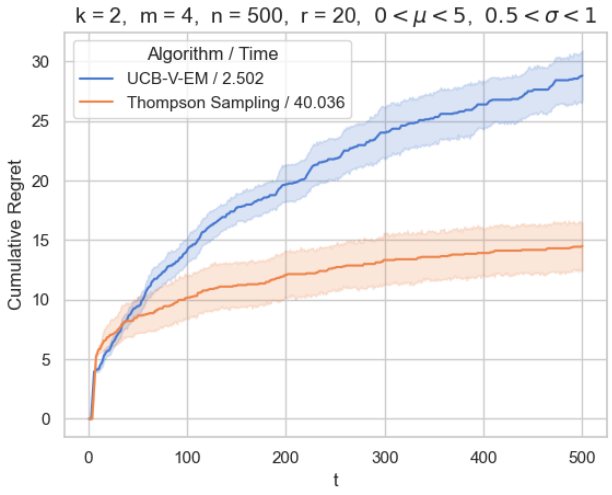


**Figure 4.5**

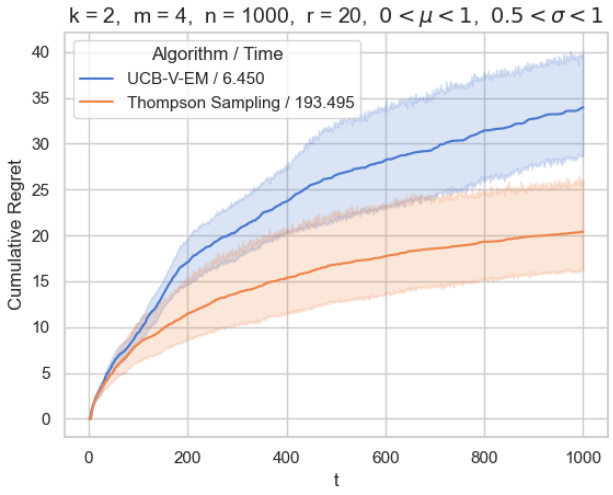


**Figure 4.6**

**4 arms case**, regret comparison for Thompson Sampling and UCB-V-EM

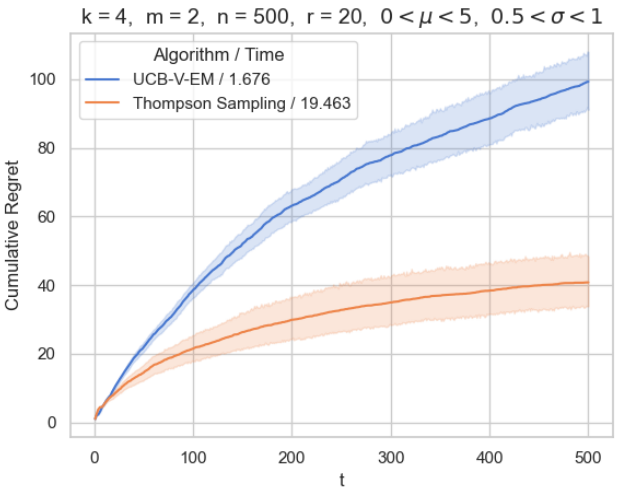


**Figure 4.7**

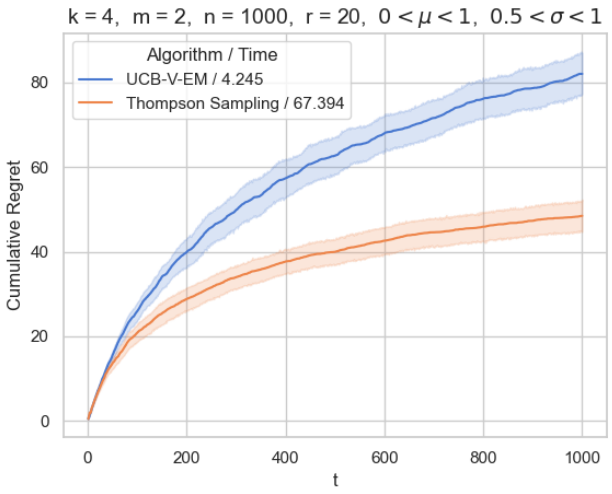


**Figure 4.8**

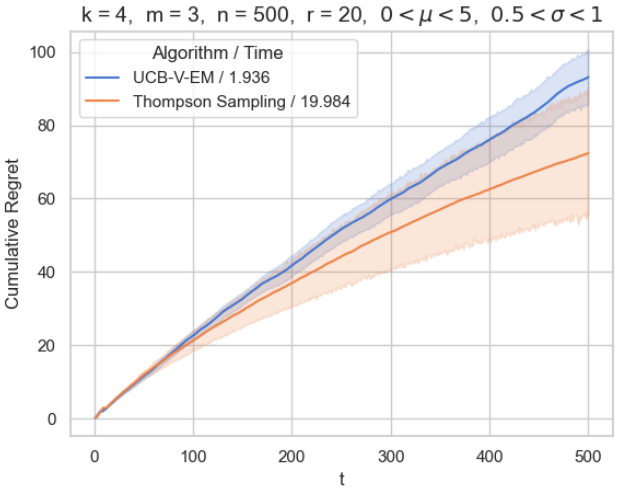


**Figure 4.9**

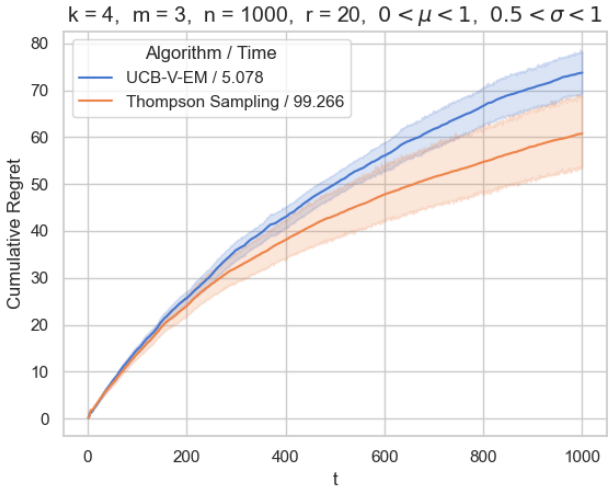


**Figure 4.10**

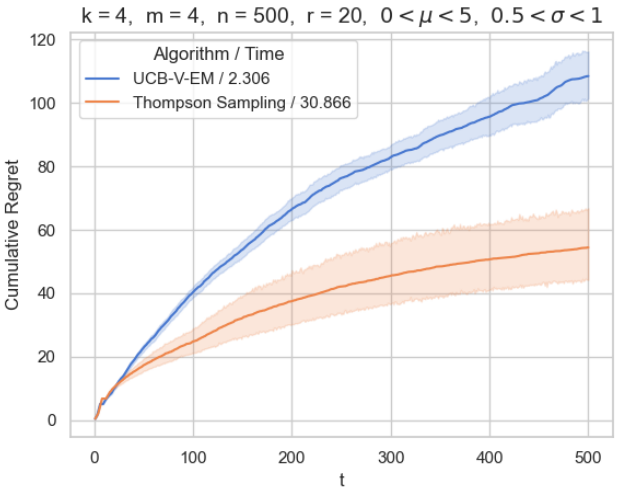


**Figure 4.11**

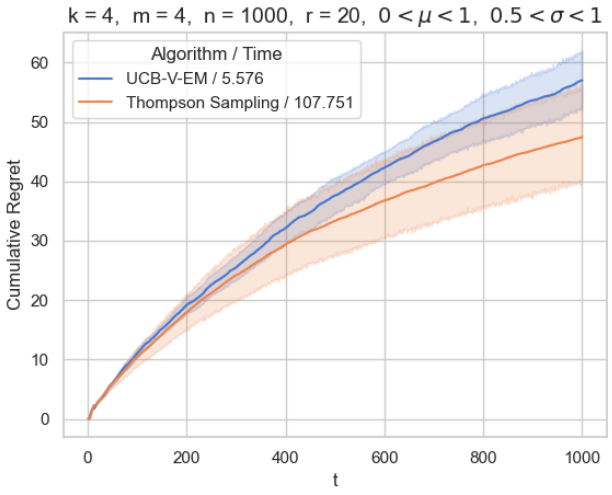


**Figure 4.12**

**8 arms case**, regret comparison for Thompson Sampling and UCB-V-EM

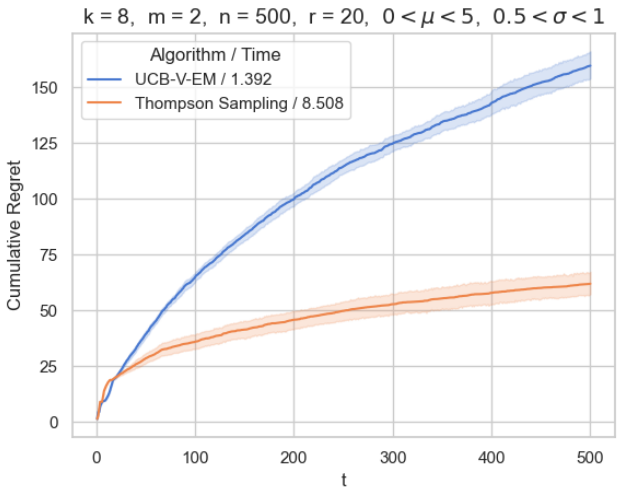


**Figure 4.13**

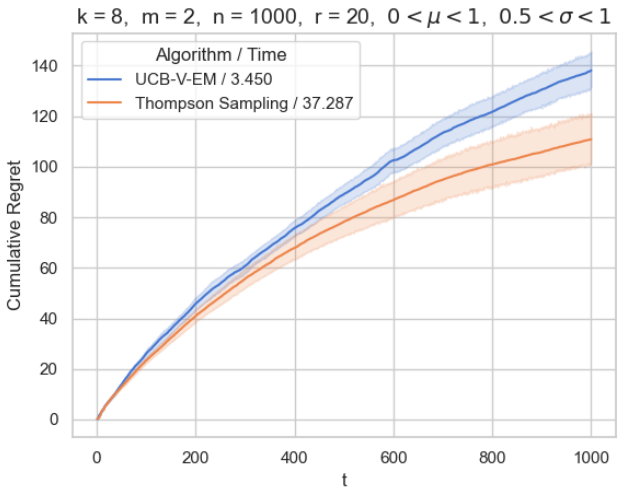


**Figure 4.14**

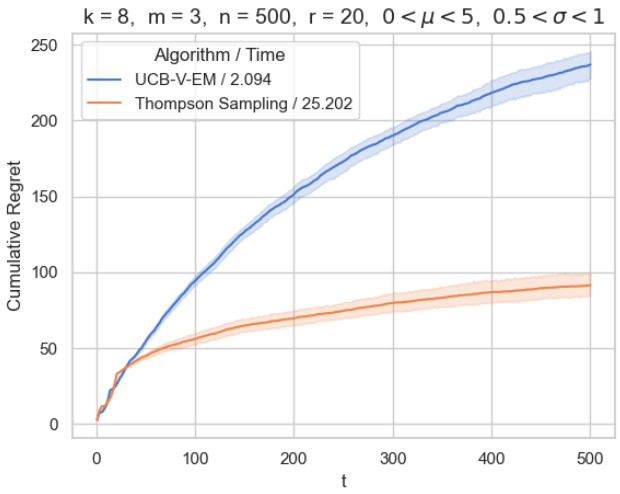


**Figure 4.15**

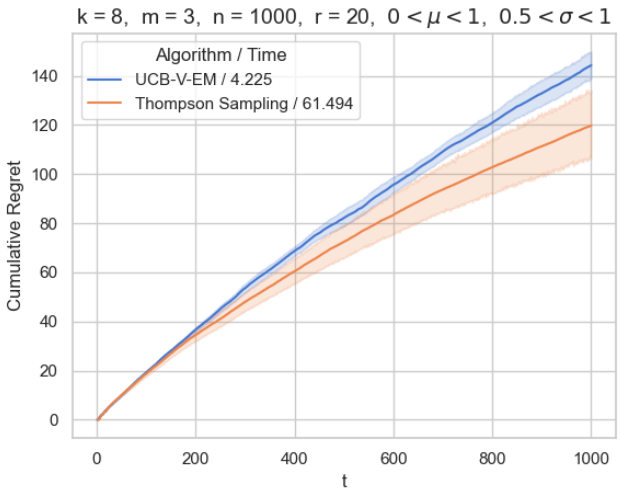


**Figure 4.16**

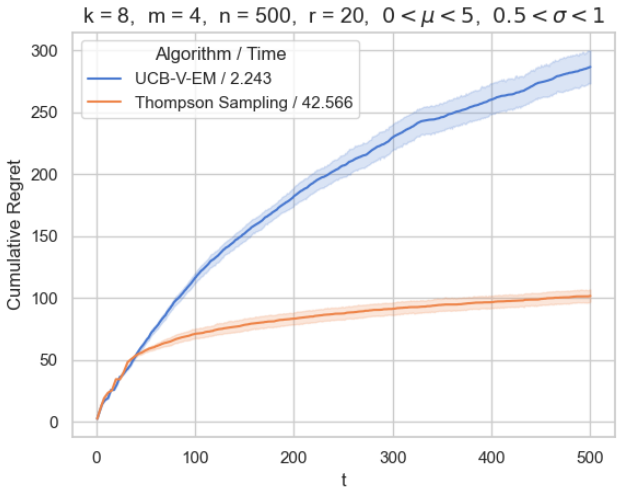


**Figure 4.17**

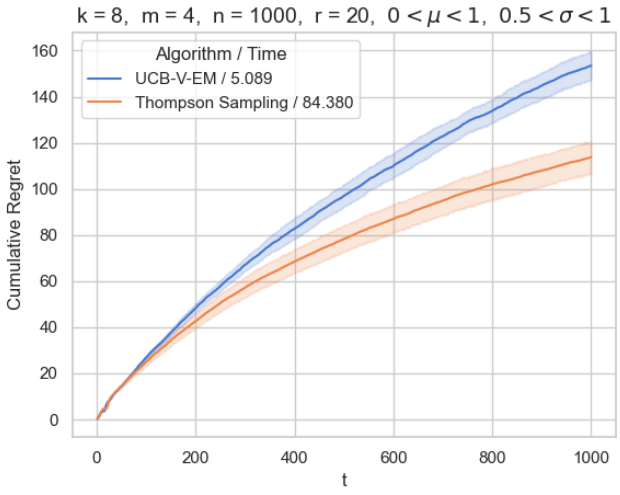


**Figure 4.18**

# 5. Mixture Model Bandits with Beta as Component Distributions

Beta distribution is also studied as another kind of reward distribution in this project. Here, the bandit environment is consisted of $k$ arms with reward that follows a mixture of independent Beta distributions. In other words, suppose the distribution of the reward of an arm is $f_X(x)$, then:

$$f_X(x) = \sum_{j=1}^{m} \pi_j \cdot b_{\alpha_j,\beta_j}(x), \qquad b_{\alpha,\beta}(x) = \frac{1}{B(\alpha,\beta)} \cdot x^{\alpha-1} \cdot (1-x)^{\beta-1},$$

where $B(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)},\ x \in [0,1]$.

The UCB-V algorithm is a main tool here because it assumes distributions over reals with bounded support $[0,b]$. The aim of this section is to see whether the performance of UCB-V algorithm can be improved with the implementation of a hybrid algorithm, which is also an iterative algorithm similar to the EM, proposed by Schröder and Rahmann (2017), to specifically deal with the problem of parameter estimations for the Beta mixture model.

The reasons why it's not recommended to directly use EM are given here. One of the main reasons why the EM algorithm is extensively used in practice for parameter estimation for mixture models is the existence of the log-likelihood. By Jensen's inequality, the likelihood increases in each EM iteration, and when it eventually stops increasing, it means the algorithm has found a stationary point. However, this solution might only be locally optimal. Global optimal solution can be found by repeating the run for several times and then comparing the log-likelihood of these stationary points.

In beta mixtures, several obstacles exist with the EM algorithm. First, the weights $W_{i,j}$ are not properly defined for $x_i = 0$ or $x_i = 1$ for some beta distribution (specifically, if one of $\alpha$ and $\beta$ is less or equal to 1). Then, the maximization step cannot be carried out if the data contains any 0 or 1. Also, even if all $x_i \in (0,1)$, the resulting mixtures are highly sensitive to perturbation of the data. Finally, for the EM algorithm for Beta mixtures, each M-step itself requires an iterative optimization procedure, the computational efforts for several EM runs are a great burden. E-MM algorithm, a hybrid algorithm that combines EM and method of moment proposed by Schröder and Rahmann (2017), is implemented in the bandit algorithm to work out the mixture model parameters in

this project.

## 5.1. Introduction to the E-MM Algorithm

E-MM algorithm is named so because it is a hybrid of the EM algorithm and the method of moments.

The algorithm proceeds iteratively as in the EM framework and alternates between EM's E step and Pearson's method of moments until the algorithm reaches convergence within a set threshold (Pearson, 1894).

To estimate $Q$ free parameters, the method of moments calculates $Q$ moments of the target distribution by expressing them in terms of the parameters, and equate them to the $Q$ sample moments. This boils down to find the solution to a system of $Q$ non-linear equations.

Thus, to avert the aforementioned problems specific to Beta mixtures, the algorithm substitute the M-step in EM by a method of moments step (MM-step). It therefore blends the idea of latent weights from EM algorithm with a method of moments estimation. It also avoids the exorbitant computational efforts from using pure moment based estimation for mixture models.

### 5.1.1. The Method of Moments for Beta Distribution

The beta distribution with parameters $\alpha > 0$ and $\beta > 0$ is a continuous probability distribution on the unit interval $[0, 1]$.

If $X$ is a random variable with a beta distribution, then its expected value $\mu$ and variance $\sigma^2$ are:

$$\mu := \mathbb{E}[X] = \frac{\alpha}{\alpha + \beta}, \quad \sigma^2 := \mathrm{Var}[X] = \frac{\mu(1-\mu)}{\alpha + \beta + 1} = \frac{\mu(1-\mu)}{1 + \phi}.$$

To use the method of moments, conversely, the parameters $\alpha$ and $\beta$ may be expressed in $\mu$ and $\sigma^2$:

$$\phi = \frac{\mu(1-\mu)}{\sigma^2} - 1; \quad \text{then} \quad \alpha = \mu\phi, \quad \beta = (1-\mu)\phi. \tag{5.1}$$

### 5.1.2. The Algorithm

This section contains the details of the E-MM algorithm proposed by Schröder and Rahmann (2017).

**Initialization**
The algorithm adapts the $D^2$-weighted initialization stage originally used in the famous k-means algorithm (Johnson, Wichern, et al., 2014). Denote $X \subset [0,1]$ to be the set of samples. Let $Y \subset X$ be the set of initial component centres, such that $Y = \{\}$ in the beginning. Let $D_Y(x) := \min_{y \in Y} |x - y|$. Then the initialization stage of the E-MM algorithm consists of:

- Sample the first point $y$ randomly and uniformly from$X$ to form $Y := \{y\}$.
- Repeat until $|Y| = c$ : Choose $y \in X \backslash Y$ with a categorical probability proportional to $D_Y(y)^2$; then set $Y := Y \cup \{y\}$.
- Sort $Y$ such that $y_1 < \cdots < y_c$
- Expectation and variance of component $j = 1, \ldots, c$ are initially estimated from the corresponding sample moments of all data points in the interval $[y_j - 0.5, y_j + 0.5]$

**E-step**
The E-step contains a minor modification from the expectation stage in EM in that the algorithm set weights of data points $x_i = 0$ and $x_i = 1$ in way that would not result in singularities.

Let $j_0$ be the component index $j$ with the smallest $\alpha_j$. If there is more than one, choose the one with the largest $\beta_j$. Then, $W_{i,j_0} = 1$ and $W_{i,j} = 0$ for $j \neq j_0$. Similarly, let $j_1$ be the component index $j$ with the smallest $\beta_j$. For all $i$ with $x_i = 1$, set $W_{i,j_1} = 1$ and $W_{i,j} = 0$ for $j \neq j_1$.

**MM-step**
The MM-step estimates mean and variance of each component $j$ by responsibility-weighted sample moments,

$$\mu_j = \frac{\sum_{i=1}^n W_{ij} \cdot x_i}{\sum_{i=1}^n W_{ij}} = \frac{\sum_{i=1}^n W_{ij} \cdot x_i}{n \cdot \pi_j}, \quad \sigma_j^2 = \frac{\sum_{i=1}^n W_{ij} \cdot (x_i - \mu_j)^2}{n \cdot \pi_j} \tag{5.2}$$

Then $\alpha_j$ and $\beta_j$ are by the method of moments as specified in Eq. 5.1 and new weights through Bayes rules.

$$W_{i,j} = \frac{\pi_j b_{\alpha_j,\beta_j}(x_i)}{\sum_k \pi_k b_{\alpha_k,\beta_k}(x_i)} \quad \text{and} \quad \pi_j^+ = \frac{1}{n}\sum_{i=1}^n W_{i,j}. \tag{5.3}$$

**Termination**
Let $\theta_q$ be any real-valued parameter to be estimated and $T_q$ a given threshold for $\theta_q$. After each MM-step, we compare $\theta_q$ (old value) and $\theta_q^+$ (updated value) by the relative change $\kappa_q := \left|\theta_q^+ - \theta_q\right| / \max\left(\left|\theta_q^+\right|, |\theta_q|\right)$. (If $\theta_q^+ = \theta_q = 0$, we set $\kappa_q := 0$.) We say that $\theta_q$ is stationary if $\kappa_q < T_q$. The algorithm terminates when all parameters are stationary.

### 5.1.3. Empirical Results for E-MM Algorithm*

Below are the empirical results that show the algorithm's effectiveness at approximating Beta mixtures.

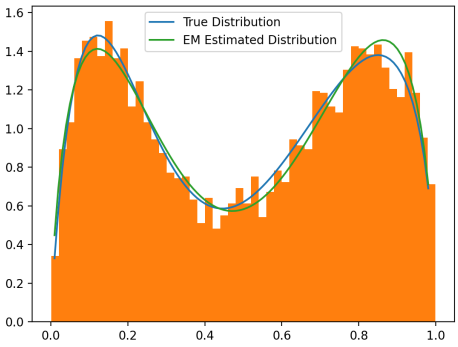


**Figure 5.1:** True distribution vs. The estimated distribution from E-MM algorithm

| | component 1 | component 2 |
|---|---|---|
| $\alpha$ | 1.92 | 4.23 |
| $\alpha^*$ | 1.80 | 4.59 |
| $\beta$ | 7.68 | 1.57 |
| $\beta^*$ | 6.82 | 2.43 |
| $\pi$ | 0.42 | 0.58 |
| $\pi^*$ | 0.50 | 0.55 |

**Table 5.1.:** Theoretical and Estimated Parameters of Mixture Beta Distribution

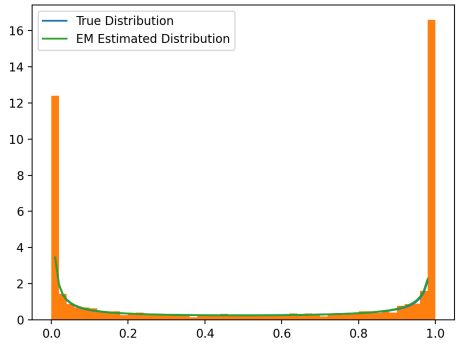


**Figure 5.2:** True distribution vs. The estimated distribution from E-MM algorithm

| | component 1 | component 2 |
|---|---|---|
| $\alpha$ | 1.68 | 0.15 |
| $\alpha^*$ | 1.81 | 0.09 |
| $\beta$ | 1.43 | 0.13 |
| $\beta^*$ | 1.68 | 0.14 |
| $\pi$ | 0.03 | 0.97 |
| $\pi^*$ | 0.11 | 0.86 |

**Table 5.2.:** Theoretical and Estimated Parameters of Mixture Beta Distribution

## 5.2. Implementation of the E-MM Algorithm in the UCB-V*

UCB-V algorithm (Audibert, Munos, and Szepesvári, 2007) is implemented to study whether the proposed E-MM algorithm is able to improve the performance.

Again, the goal here is to use E-MM algorithm to obtain a hopefully better estimate for the empirical variance. For any $k \in \{1, \ldots, K\}$ and $t \in \mathbb{N}$, let $\bar{X}_{k,t}$ and $V_{k,t}$ be the

empirical estimates of the mean payoff and variance of arm $k$, then:

$$\bar{X}_{k,t} \triangleq \frac{1}{t}\sum_{i=1}^{t} X_{k,i} \quad \text{and} \quad V_{k,t} \triangleq \frac{1}{t}\sum_{i=1}^{t}\left(X_{k,i} - \bar{X}_{k,t}\right)^2$$

If we use E-MM Algorithm to estimate the variance term, as derived in Eq. 3.5, it can be estimated as:

$$\bar{V}_{k,t} \triangleq \sum_{i=1}^{m} \hat{\pi}_i \left(\hat{\sigma}_i^2 + \hat{\mu}_i^2\right) - \hat{\mu}^2,$$

If the mixture model assumption is ignored, then the UCB-V algorithm seeks to maximize the bound in Eq. 3.3 in each play:

$$B_{k,s,t} \triangleq \bar{X}_{k,s} + \sqrt{\frac{2V_{k,s}\log t}{s}} + \frac{b\log t}{2s},$$

where $V_{k,s}$ is the simple sample variance as written above, $b$ is the supremum of the reward, $k$ the arm, $s$ the number of samples from arm $k$ until time $t$. Note that $b = 1$ in the Beta mixture case and there is no need to estimate it, because the support of the Beta distribution is on $[0, 1]$.

If the mixture model structure is considered, then use the E-MM algorithm to estimate $V_{k,s}$ by the mixture model variance specified in Eq. 3.5.

## 5.3. Empirical Results for Beta Mixture Model Bandits*

Even though the E-MM algorithm works quite well for estimating parameters of single Beta mixture model, the algorithms doesn't seem to have improved the performance of the UCB-V algorithm significantly as indicated in the regret plots presented afterwards.

The algorithm is arguably more complicated than EM and therefore takes more computational efforts as expected. It increases running time by the scale or nearly 1000 folds but doesn't pay off with a significant improvement in the cumulative regret.

Unfortunately, the bandit algorithm itself also doesn't work quite well here. Note that in most of the plots (especially those in the 4-components case) the algorithm doesn't seem to have achieved logarithmic regret in the given horizon. This could be due to the fact that a mixture of Beta distributions (which can have a big variety of shapes with different values of $\alpha$ and $\beta$) might be too challenging for the bandit algorithm to learn quickly.

Regret Plots for UCB-V-EM and UCB-V for Beta Mixture, **2** Components

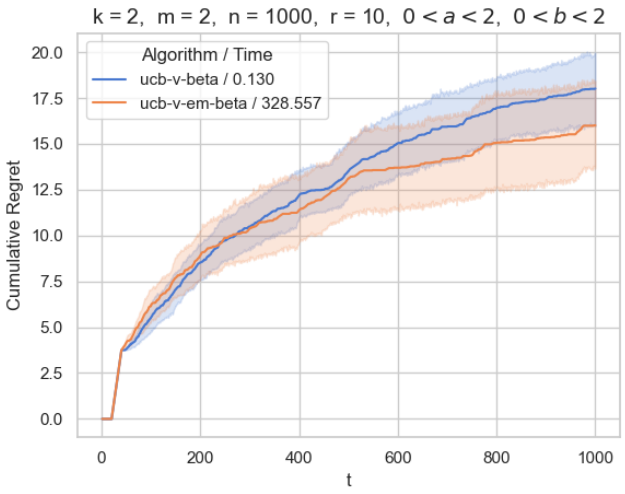


**Figure 5.3**

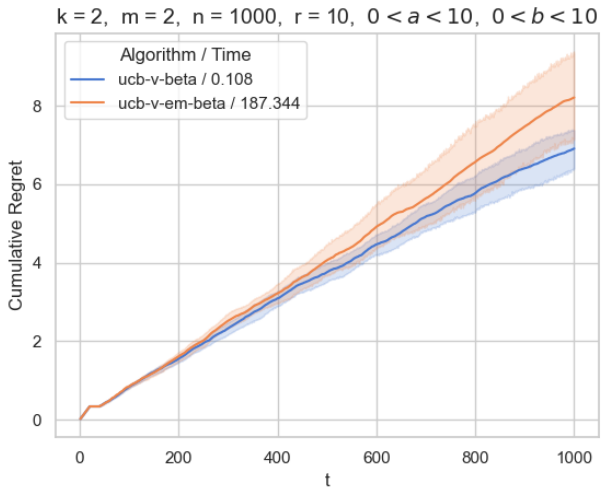


**Figure 5.4**

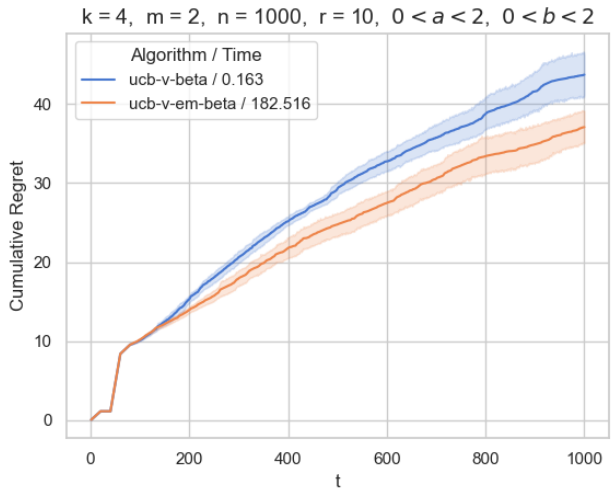


**Figure 5.5**

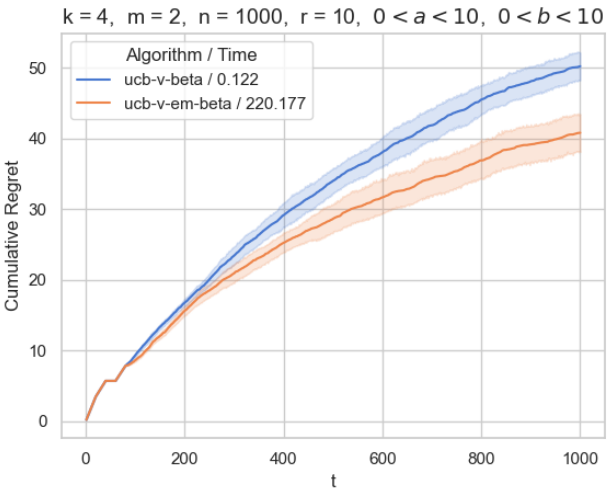


**Figure 5.6**

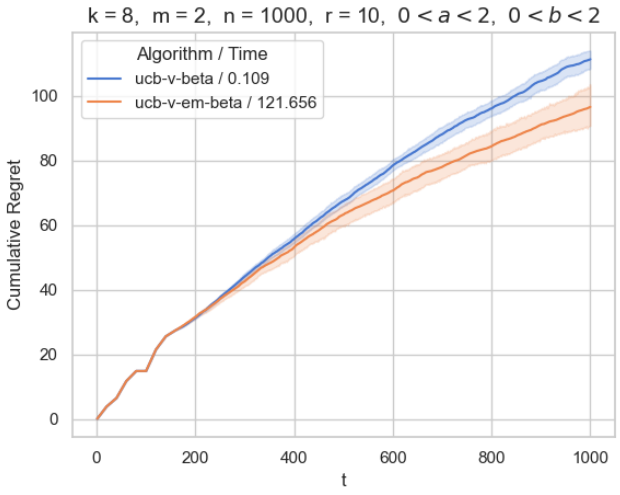


**Figure 5.7**

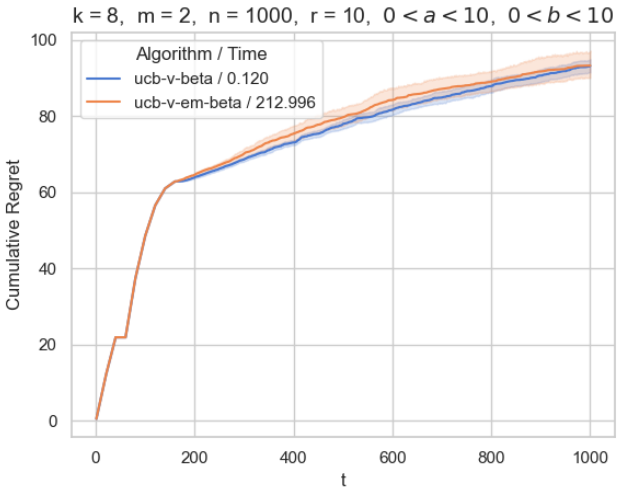


**Figure 5.8**

Regret Plots for UCB-V-EM and UCB-V for Beta Mixture, **3** Components

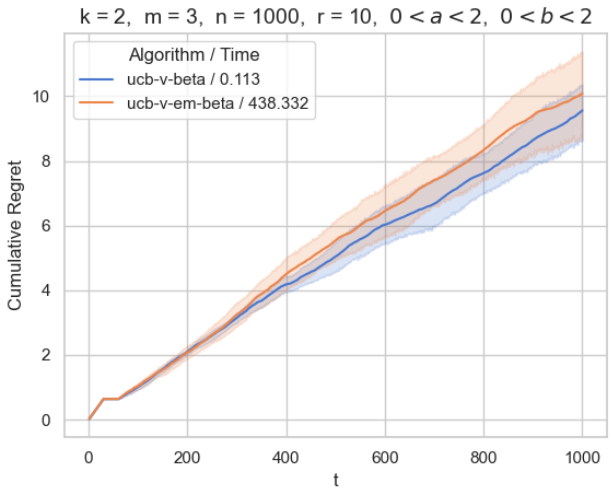


**Figure 5.9**

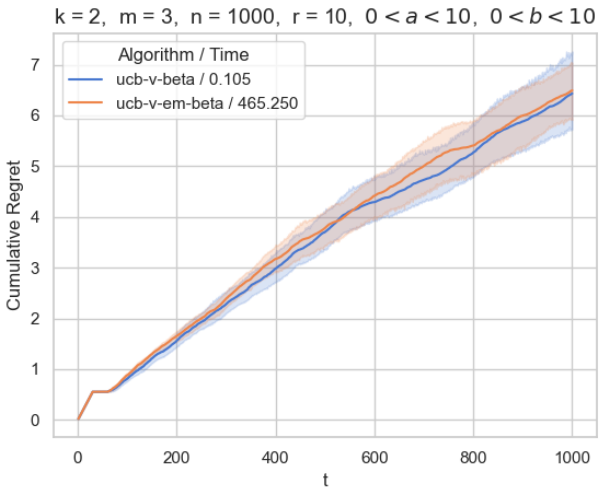


**Figure 5.10**

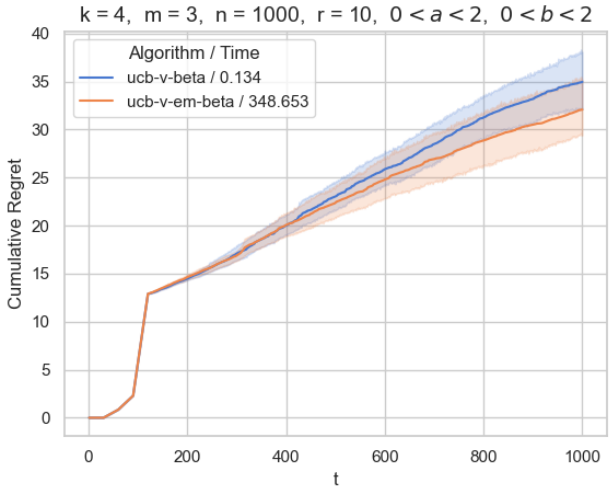


**Figure 5.11**

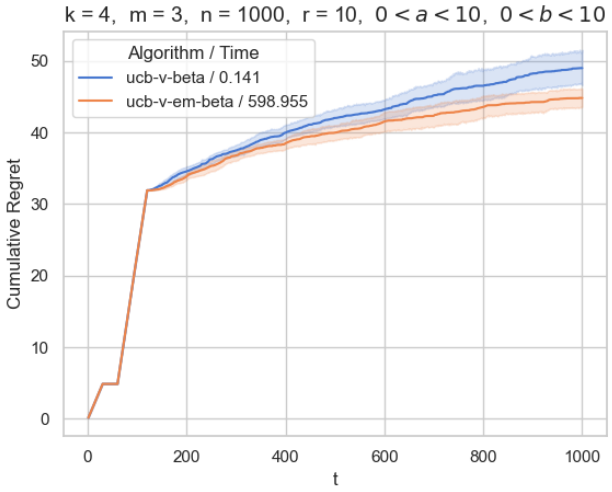


**Figure 5.12**

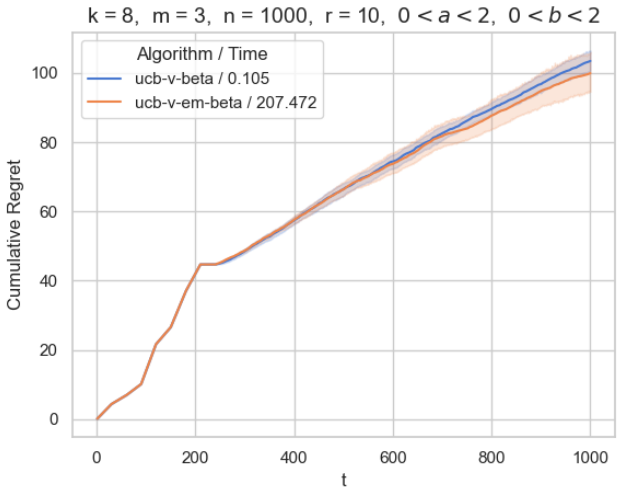


**Figure 5.13**

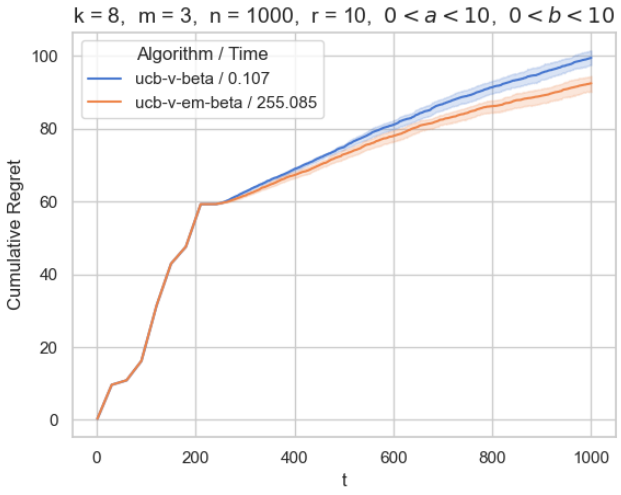


**Figure 5.14**

# 6. Conclusion*

In the known variance case, experiments have been done for UCB and UCB-EM algorithm for mixture model bandit problem. The improvement in the performance from EM is apparent from the regret plots. By estimating the weights of each arm, EM has provided us with a more accurate estimate or bound for the overall subguassianity of a mixture model and therefore improved the performance of the algorithm significantly.

However, EM doesn't bring us with significant benefits when the assumptions change to that the variances are totally unknown to the player. The empirical results are that the performances of the UCB-V and UCB-V-EM are almost entirely the same except that UCB-V-EM is slower because of more computational burden. This is also expected because what EM can really do here is to utilize this only new information, the number of components, to try to provide a better estimation for the overall variances. At the same time, the simple empirical sample variance is already an asymptotically unbiased estimator for the overall variance no matter we assume mixture model assumptions or not.

UCB1-NORMAL completely falls out of discussion because its performance is strictly dominated by that of the UCB-V algorithm under mixture model assumption. This is also a quite reasonable and explainable result because the assumption of UCB1-NORMAL is that the rewards are strictly Gaussian, which is not the case here.

Thompson sampling is arguably the best algorithm here in terms of sheer cumulative regret. However, it takes much more computational efforts and should not be used without considering the practical needs.

In the Beta mixture model case, the improvement by EM is not evident.

# Bibliography


[Attias et al. 1999] Attias, Hagai et al.: A Variational Baysian Framework for Graphical Models. In: *NIPS* Bd. 12 Citeseer, 1999

[Audibert et al. 2007] Audibert, Jean-Yves ; Munos, Rémi ; Szepesvári, Csaba: Tuning bandit algorithms in stochastic environments. In: *International conference on algorithmic learning theory* Springer, 2007, S. 150–165

[Auer et al. 2002] Auer, Peter ; Cesa-Bianchi, Nicolo ; Fischer, Paul: Finite-time analysis of the multiarmed bandit problem. In: *Machine learning* 47 (2002), Nr. 2, S. 235–256

[Beal 2003] Beal, Matthew J.: *Variational algorithms for approximate Bayesian inference*. University of London, University College London (United Kingdom), 2003

[Blei and Jordan 2006] Blei, David M. ; Jordan, Michael I.: Variational inference for Dirichlet process mixtures. In: *Bayesian analysis* 1 (2006), Nr. 1, S. 121–143

[Dempster et al. 1977] Dempster, Arthur P. ; Laird, Nan M. ; Rubin, Donald B.: Maximum likelihood from incomplete data via the EM algorithm. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 39 (1977), Nr. 1, S. 1–22

[Johnson et al. 2014] Johnson, Richard A. ; Wichern, Dean W. et al.: *Applied multivariate statistical analysis*. Bd. 6. Pearson London, UK:, 2014

[Lattimore and Szepesvári 2020] Lattimore, Tor ; Szepesvári, Csaba: *Bandit algorithms*. Cambridge University Press, 2020

[Pearson 1894] Pearson, Karl: Contributions to the mathematical theory of evolution. In: *Philosophical Transactions of the Royal Society of London. A* 185 (1894), S. 71–110

[Schröder and Rahmann 2017] Schröder, Christopher ; Rahmann, Sven: A hybrid parameter estimation algorithm for beta mixtures and applications to methylation state classification. In: *Algorithms for Molecular Biology* 12 (2017), Nr. 1, S. 1–12

[Thompson 1933] Thompson, William R.: On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. In: *Biometrika* 25 (1933), Nr. 3/4, S. 285–294

# A. Python Codes

## A.1. bandit_class.py

```python
import numpy as np
from scipy.stats import norm, uniform
from tabulate import tabulate
from pprint import pprint


class MixtureBandit:
    # Mixture Model Bandit class
    def __init__(self, arm_num, components_num=2, mean_range=None,
        sd_range=None):
        # arm_num: k, how many arms
        # components_num: how many components
        # mean_range: the range from which the component means are
            sampled from
        # sd_range: the range from which the component standard
            deviations are sampled from

        self.is_beta = False

        self.arm_num = arm_num
        self.components_num = components_num

        self.mean_range = mean_range
        self.sd_range = sd_range

        self.alpha = np.random.dirichlet([1]*components_num, size=arm_num
            )
        self.mean_list = np.random.uniform(low=mean_range[0], high=
            mean_range[1],size=[arm_num, components_num])
        self.sd_list = np.random.uniform(low=sd_range[0], high=sd_range
            [1],size=[arm_num, components_num])

        self.weighted_mean = [np.dot(self.alpha[i], self.mean_list[i])
            for i in range(self.arm_num)]
        self.optimal_arm_mean = max(self.weighted_mean)
        self.optimal_arm = self.weighted_mean.index(self.optimal_arm_mean
            ) + 1

    def print_optimal_arm(self):
        print(f"\nBest arm is the arm {self.optimal_arm} "
              f"with mean {self.optimal_arm_mean:1.3f}\n")

```

```
35      def get_para(self):
36          # output the parameters of the environment
37          return {'mean': self.mean_list, 'sd': self.sd_list, 'alpha': self
                .alpha}
38
39      def pull(self, arm, size):
40          # get rewards from an arm
41          arm_mean = self.mean_list[arm-1]
42          arm_sd = self.sd_list[arm-1]
43          my_sample = []
44          for i in range(size):
45              component_chosen = list(np.random.multinomial(1, self.alpha[
                    arm-1]))
46              component_chosen = component_chosen.index(max(
                    component_chosen))
47              my_sample.append(norm.rvs(loc=arm_mean[component_chosen],
                    scale=arm_sd[component_chosen],size=size)[0])
48          return my_sample
49
50      def parameters(self, latex: bool):
51          # return a tabulate that displays the parameters of a bandit
                environment
52          table = list([])
53
54          for component_index in range(self.components_num):
55              mean = ['mean '+str(component_index+1)] + list(np.round([self
                    .mean_list[i][component_index] for i in range(self.arm_num
                    )], 2))
56              sd = ['std ' + str(component_index + 1)] + list(np.round([
                    self.sd_list[i][component_index] for i in range(self.
                    arm_num)], 2))
57              alpha = ['alpha ' + str(component_index + 1)] + list(np.round
                    ([self.alpha[i][component_index] for i in range(self.
                    arm_num)], 2))
58              table.append(alpha)
59              table.append(mean)
60              table.append(sd)
61              table.append(['-']+['']*self.arm_num)
62          table.append(['weighted mean'] + list(np.round(self.weighted_mean
                , 2)))
63          headers = ['arm ' + str(x + 1) if x + 1 != self.optimal_arm
64                     else 'arm ' + str(x + 1) + '*' for x in range(self.
                         arm_num)]
65          if latex:
66              return tabulate(table, headers=headers, tablefmt='latex')
67          return tabulate(table, headers=headers, tablefmt='simple')
68
69
70  # my_bandit = MixtureBandit(arm_num=2, components_num=3, mean_range=[0,
        10], sd_range=[0.5, 1])
71  # print(my_bandit.parameters(latex=True))
72  # my_bandit.print_optimal_arm()
73  # print(np.mean(my_bandit.pull(arm=0, size=1000)))
```

## A.2. bandit_class_beta.py

```
import numpy as np
from scipy.stats import norm, uniform
from tabulate import tabulate
from pprint import pprint


class MixtureBanditBeta:
    # the Beta mixture model Bandit class, the structure is
    # mostly the same as 'MixtureBandit'
    def __init__(self, arm_num, alpha_range, beta_range, components_num
        =2):
        self.is_beta = True

        self.arm_num = arm_num
        self.components_num = components_num

        self.alpha_range = alpha_range
        self.beta_range = beta_range

        self.pi = np.random.dirichlet([1]*components_num, size=arm_num)
        self.alpha_list = np.random.uniform(low=alpha_range[0], high=
            alpha_range[1], size=[arm_num, components_num])
        self.beta_list = np.random.uniform(low=beta_range[0], high=
            beta_range[1], size=[arm_num, components_num])

        self.mean_list = self.alpha_list/(self.alpha_list+self.beta_list)
        self.var_list = self.alpha_list*self.beta_list/\
                        (self.alpha_list+self.beta_list)**2\
                        /(self.alpha_list+self.beta_list+1)

        self.weighted_mean = [np.dot(self.mean_list[i], self.pi[i]) for i
             in range(self.arm_num)]
        self.optimal_arm_mean = max(self.weighted_mean)
        self.optimal_arm = self.weighted_mean.index(self.optimal_arm_mean
            ) + 1

    def print_optimal_arm(self):
        print(f"\nBest arm is the arm {self.optimal_arm} "
              f"with mean {self.optimal_arm_mean:1.3f}\n")

    def get_para(self):
        return {'alpha': self.alpha_list, 'beta': self.beta_list, 'pi':
            self.pi}

    def pull(self, arm, size):
        arm_alpha = self.alpha_list[arm-1]
        arm_beta = self.beta_list[arm-1]
        my_sample = []
        for i in range(size):
            component_chosen = list(np.random.multinomial(1, self.pi[arm
                -1]))
```

```
            component_chosen = component_chosen.index(max(
                component_chosen))
            my_sample.append(np.random.beta(a=arm_alpha[component_chosen
                ], b=arm_beta[component_chosen], size=size)[0])
        return my_sample

    def parameters(self, latex: bool):
        table = list([])

        for component_index in range(self.components_num):
            alpha = ['alpha '+str(component_index+1)] + \
                    list(np.round([self.alpha_list[i][component_index]
                                   for i in range(self.arm_num)], 2))
            beta = ['beta ' + str(component_index + 1)] + \
                   list(np.round([self.beta_list[i][component_index]
                                  for i in range(self.arm_num)], 2))
            pi = ['pi ' + str(component_index + 1)] + \
                 list(np.round([self.pi[i][component_index]
                                for i in range(self.arm_num)], 2))
            table.append(alpha)
            table.append(beta)
            table.append(pi)
            table.append(['-']+['']*self.arm_num)
        table.append(['weighted mean'] + list(np.round(self.weighted_mean
            , 2)))
        headers = ['arm ' + str(x + 1) if x + 1 != self.optimal_arm
                   else 'arm ' + str(x + 1) + '*' for x in range(self.
                       arm_num)]
        if latex:
            return tabulate(table, headers=headers, tablefmt='latex')
        return tabulate(table, headers=headers, tablefmt='simple')


# my_bandit = MixtureBanditBeta(arm_num=3, components_num=2, alpha_range
    =[0, 5], beta_range=[0, 5])
# print(my_bandit.parameters(latex=False))
# my_bandit.print_optimal_arm()
# print(my_bandit.pull(arm=1, size=10))
#
```

## A.3. UCB_known_sd.py

```
import numpy as np
from scipy.stats import norm, uniform
from sklearn.cluster import KMeans
from bandit_class import MixtureBandit


def ucb_bound(t, prev_t, mu, sd, alpha_value):
    # This is the upper confidence bound for unknown variance case
    # input: parameters for the bound
    # output: the UCB bound
    def ft(x): return 1 + x * np.log(x) ** 2
```

```
    if alpha_value is not None:
        var_est = np.dot(alpha_value, sd ** 2)
        return mu + np.sqrt(2 * var_est * np.log(ft(t)) / prev_t)
    return mu + np.sqrt(2 * np.max(sd ** 2) * np.log(ft(t)) / prev_t)


def em_known_var(x, sd, m, acc, prev_mean, prev_alpha):
    # EM algorithm for known variance case
    # INPUT:
    # x: samples,
    # sd: standard deviations,
    # m: components number
    # acc: accuracy criterion for convergence
    # prev_mean: component means when the algorithm converged last time
    # prev_alpha: weights when the algorithm converged last time

    # OUTPUT:
    # [component means, component weights]
    x = np.array(x)
    big_n = len(x)
    if (np.min(prev_alpha) > 0.01) & (np.min(1 - prev_alpha) > 0.01) & \
            (np.min(np.abs(prev_alpha - 1/m)) > 0.01) & (big_n > 50):
        mean = prev_mean
        alpha = prev_alpha
    else:
        k_means = KMeans(n_clusters=m, random_state=0).fit(x.reshape(-1,
            1))
        mean = np.squeeze(k_means.cluster_centers_)
        alpha = uniform.rvs(loc=0.4, scale=0.3, size=m)

    while True:
        p = np.array([norm.pdf(x, loc=mean[i], scale=sd[i]) for i in
            range(m)])
        p_x = np.array([np.dot([p[j][i] for j in range(m)], alpha) for i
            in range(big_n)])
        w = np.array([alpha[i] * p[i] / p_x for i in range(m)])

        new_alpha = np.array([np.sum(w[i]) for i in range(m)]) / big_n
        new_mean = np.array([1 / np.sum(w[i]) * np.dot(w[i], x) for i in
            range(m)])

        if (np.max(np.abs(mean - new_mean)) < acc) & (np.max(np.abs(
            new_alpha - alpha)) < acc):
            break
        alpha = new_alpha
        mean = new_mean
    return [mean, alpha]


def ucb(bandit: MixtureBandit, n: int):
    # UCB algorithm for the known variance case
    # INPUT:
    # bandit: a bandit instance from MixtureBandit class
```

```
    # n: horizon

    # OUTPUT:
    # [[action 1, reward 1],...,[action n, reward n]]
    k = bandit.arm_num  # how many arms?
    m = bandit.components_num
    samples = []
    mu_lst = np.array([0.] * k)  # means of each arm
    sd = np.array(bandit.sd_list)
    count_lst = [0] * k

    size = np.max([m, 10])
    for i in range(k):
        sample = bandit.pull(arm=i + 1, size=size)
        for j in range(size):
            samples.append([i + 1, sample[j]])
        count_lst[i] += size
        mu_lst[i] = np.mean(sample)

    for t in range(k * size, n):
        if t % 100 == 0:
            print('t = ', t)
        ucb_lst = [ucb_bound(t=t,
                             prev_t=count_lst[j],
                             mu=mu_lst[j],
                             sd=sd[j],
                             alpha_value=None) for j in range(k)]

        action = ucb_lst.index(max(ucb_lst)) + 1
        count_lst[action - 1] += 1
        new_x = bandit.pull(arm=action, size=1)
        samples.append([action, new_x])
        mu_lst[action - 1] = np.mean([x[1] for x in samples if x[0] ==
            action])
    print(count_lst)
    return samples


def ucb_em(bandit: MixtureBandit, n: int):
    # UCB-EM Algorithm for the known variance case
    # INPUT
    # bandit: a bandit instance from the MixtureBandit class
    # n: horizon

    # OUTPUT
    # [[action 1, reward 1],...,[action n, reward n]]
    print(bandit.parameters(latex=False))
    k = bandit.arm_num
    m = bandit.components_num
    samples = []
    mu_lst = np.array([0.] * k)
    sd = bandit.sd_list

```

```
    prev_means = np.array([[0.] * m] * k)
    prev_alphas = np.random.dirichlet([1]*m, size=k)
    count_lst = [0] * k

    size = np.max([10, m])
    for i in range(k):
        sample = bandit.pull(arm=i + 1, size=size)
        count_lst[i] += size
        for j in range(size):
            samples.append([i + 1, sample[j]])
        mu_lst[i] = np.mean(sample)
        prev_means[i] = [np.mean(sample)]*m

    action = 0
    ucb_lst_em = 0
    for t in range(k * size, n):  # do the UCB algorithm for remaining of
        t from k+1 to n
        if t % 100 == 0:
            print(f"t = {t}, chosen arm: {action}, ucb_em: {ucb_lst_em}
                count_lst: {count_lst}")

        ucb_lst_em = [ucb_bound(t=t, prev_t=count_lst[j], mu=mu_lst[j],
            sd=sd[j],
                                alpha_value=prev_alphas[j]) for j in
                                    range(k)]

        action = ucb_lst_em.index(max(ucb_lst_em)) + 1
        new_x = bandit.pull(arm=action, size=1)[0]
        samples.append([action, new_x])
        count_lst[action-1] += 1

        prev_samples = [x[1] for x in samples if x[0] == action]
        [means, alpha] = em_known_var(x=prev_samples,
                                      sd=sd[action - 1],
                                      m=m,
                                      acc=1e-3,
                                      prev_mean=prev_means[action - 1],
                                      prev_alpha=prev_alphas[action - 1])
        prev_means[action - 1] = means
        prev_alphas[action - 1] = alpha
        mu_lst[action - 1] = np.mean([x[1] for x in samples if x[0] ==
            action])
    print(count_lst)
    return samples
```

## A.4. UCB_Normal.py

```
import numpy as np
from UCB_known_sd import *
from UCB_V import gm_sol


def ucb_normal(bandit: MixtureBandit, n: int, em=False):
```

```
    # UCB-NORMAL algorithm
    # INPUT:
    # bandit: a bandit instance from the MixtureBandit class
    # n: horizon
    # em: boolean, use EM or not?
    # OUTPUT:
    # [[action 1, reward 1],...,[action n, reward n]]
    k = bandit.arm_num
    samples = []
    count_lst = [0] * k
    sd_lst = [1] * k
    bounds = 0
    for t in range(1, n+1):
        if t % 200 == 0:
            print(f"t = {t}, bounds = {bounds}, count = {count_lst}")
        check_log = count_lst < np.ceil(8 * np.log(t))
        if t == 1:
            check_log[0] = True
        cond = False
        for i in range(1, k+1):
            if check_log[i-1]:
                samples.append([i, bandit.pull(arm=i, size=1)[0]])
                count_lst[i-1] += 1
                cond = True
                break
        if cond:
            continue

        bounds = [0]*k
        for arm in range(1, k+1):
            prev_samples = np.array([x[1] for x in samples if x[0] == arm
                ])
            n = len(prev_samples)
            x_mean = np.mean(prev_samples)

            if em:
                bounds[arm - 1] = x_mean + np.sqrt(16 * np.square(sd_lst[
                    arm-1]) * np.log(t - 1) / n)
            else:
                q_j = np.sum(np.square(prev_samples))
                bounds[arm - 1] = x_mean + np.sqrt(16 * q_j/(n - 1) * np.
                    log(t-1)/n)

        action = bounds.index(max(bounds)) + 1
        new_x = bandit.pull(arm=action, size=1)[0]
        if em:
            prev_samples = np.array([x[1] for x in samples if x[0] ==
                action])
            solutions = gm_sol(prev_samples, bandit.components_num)
            sd_lst[action-1] = solutions['sd']
        samples.append([action, new_x])
        count_lst[action - 1] += 1
    print(count_lst)
```

```
    return samples


def ucb_normal_em(bandit, n):
    return ucb_normal(bandit, n, em=True)
```

## A.5. UCB_V.py

```
import numpy as np
from bandit_class import MixtureBandit
from sklearn.mixture import GaussianMixture
from UCB_known_sd import *


def gm_sol(x, m):
    # Function for solving gaussian mixture parameters
    # INPUT:
    # x: samples from an arm
    # m: how many components?
    # OUTPUT:
    # sd: estimated overall standard deviation of the arm
    x = np.array(x)
    x_bar = np.mean(x)
    gm = GaussianMixture(n_components=m, random_state=0, warm_start=True)
        .fit(x.reshape(-1, 1))
    component_means = gm.means_.reshape(m)
    component_vars = gm.covariances_.reshape(m)
    my_cov = np.dot(component_means**2+component_vars, gm.weights_) -
        x_bar**2
    return {'sd': np.sqrt(my_cov)}


def ucb_bern_bound(mu, var, b, t, big_t):
    # UCB bernstein bound
    return mu + np.sqrt(2 * var * np.log(t) / big_t) + b * np.log(t) /
        (2*big_t)


def ucb_v(bandit: MixtureBandit, n: int, em=False):
    # UCB-V Algorithm
    # bandit: a bandit instance from the MixtureBandit class
    # n: horizon
    # em: use EM algorithm or not?
    # OUTPUT:
    # [[action 1, reward 1],...,[action n, reward n]]
    k = bandit.arm_num
    m = bandit.components_num
    samples = []
    mu_lst = [0] * k
    sd_lst = [0] * k
    count_lst = [0] * k
    b = 1

```

```
    for i in range(k):
        initial_samples = bandit.pull(arm=i+1, size=m-1)
        if max(initial_samples) > b:
            b = max(initial_samples)
        for sample in initial_samples:
            samples.append([i+1, sample])
        mu_lst[i] = np.mean(initial_samples)
        sd_lst[i] = 1
        count_lst[i] += m-1

    ucb_lst = 0
    for t in range(k*(m-1), n):
        if t % 200 == 0:
            print(f"t = {t}, bernstein bounds = {ucb_lst}, count = {
                count_lst}")

        ucb_lst = [ucb_bern_bound(t=t, big_t=count_lst[i],
                                  mu=mu_lst[i],
                                  var=sd_lst[i] ** 2,
                                  b=b) for i in range(k)]
        action = ucb_lst.index(max(ucb_lst)) + 1
        new_x = bandit.pull(arm=action, size=1)[0]
        if new_x > b:
            b = new_x
        samples.append([action, new_x])
        count_lst[action-1] += 1

        prev_samples = [x[1] for x in samples if x[0] == action]
        mu_lst[action-1] = np.mean(prev_samples)
        if not em:
            sd_lst[action-1] = np.std(prev_samples)
        else:
            solution = gm_sol(prev_samples, m)
            sd_lst[action-1] = solution['sd']
    print(count_lst, ', b = ', b)
    return samples


def ucb_v_em(bandit, n):
    return ucb_v(bandit, n, em=True)
```

## A.6. Thompson_Sampling.py

```
from sklearn.mixture import BayesianGaussianMixture
from bandit_class import MixtureBandit
from scipy.stats import norm
from pprint import pprint
import numpy as np


def posterior_para(sample, bandit):
    # Posterior of the parameters
    # INPUT:
```

```
    # sample: samples
    # bandit: a bandit instance from the bandit class
    # OUTPUT:
    # a dictionary containing:
    # 'dirichlet_concentration': concentration parameters of the posterior
    #                            dirichlet distribution for the weights
    # 'mean': means of the posterior of component mean
    # 'mean_sd': standard deviations of the posterior of component mean
    x = np.squeeze(np.array(sample)).reshape(-1, 1)
    bgm = BayesianGaussianMixture(n_components=bandit.components_num,
                                  mean_precision_prior=0.1,
                                  covariance_prior=[[np.mean(bandit.sd_range)**2]],
                                  max_iter=5000,
                                  weight_concentration_prior_type='dirichlet_distribution',
                                  warm_start=True).fit(x)

    return {'dirichlet_concentration': np.squeeze(bgm.weight_concentration_),
            'mean': np.squeeze(bgm.means_),
            'mean_sd': 1/np.sqrt(bgm.mean_precision_)}


def thompson_sampling(bandit: MixtureBandit, n: int):
    # Thompson sampling algorithm
    # INPUT:
    # bandit: a bandit instance from the bandit class
    # n: horizon
    # OUTPUT:
    # [[action 1, reward 1],...,[action n, reward n]]
    k = bandit.arm_num
    m = bandit.components_num
    para_list = [0]*k
    samples = []
    count_lst = [0]*k
    for i in range(k):
        first_samples = bandit.pull(arm=i+1, size=m)
        para_list[i] = posterior_para(first_samples, bandit)
        for j in range(m):
            samples.append([i + 1, first_samples[j]])
        count_lst[i] += m

    for i in range(m*k, n):
        if i % 200 == 0:
            print('t = ', i)
            # pprint(para_list, indent=2)
        mean_list = [0]*k
        for j in range(k):
            para_arm_j = para_list[j]

            means = para_arm_j['mean']
```

```
            mean_sds = para_arm_j['mean_sd']
            dirichlet_concentration = para_arm_j['dirichlet_concentration
                ']

            mean_posterior_list = [float(norm.rvs(loc=means[i], scale=
                mean_sds[i], size=1)) for i in range(m)]
            dirichlet_sample = np.squeeze(np.random.dirichlet(
                dirichlet_concentration, size=1))

            weighted_mean = np.dot(mean_posterior_list, dirichlet_sample)
            mean_list[j] = weighted_mean

        action = mean_list.index(max(mean_list))+1
        count_lst[action-1] += 1
        new_x = bandit.pull(arm=action, size=1)[0]
        samples.append([action, new_x])

        prev_samples = np.array([x[1] for x in samples if x[0] == action
            ])
        para_list[action-1] = posterior_para(prev_samples, bandit)

    print(count_lst)
    for dic in para_list:
        for key, value in dic.items():
            print(f'{key:25}{value}')
    return samples
```

## A.7. UCB_V_Beta.py

```
import numpy as np
import matplotlib.pyplot as plt
from bandit_class_beta import MixtureBanditBeta
from UCB_known_sd import *
from scipy.stats import beta
from scipy.interpolate import *
from pprint import pprint
from regret_comparison import *


def check_cond(new_para, prev_para, threshold):
    # function that checks whether the algorithm has converged
    # new_para: the new set of parameters
    # prev_para: the previous set of parameters
    # threshold: the threshold

    # OUTPUT:
    # Boolean, TRUE or FALSE

    if new_para == prev_para or np.max([np.abs(new_para), np.abs(
        prev_para)]) == 0:
        return True
    Kq = np.abs(new_para-prev_para) / np.max([np.abs(new_para), np.abs(
        prev_para)])
```

```
    if Kq < threshold:
        return True
    else:
        return False


def em_beta(x, m, acc, cache_para=None):
    # the EM-M algorithm for solving parameters of a beta mixture
    # x: samples
    # m: components number
    # acc: accuracy for convergence
    # cache_para: previous converged parameters when the algorithm ran
        last time

    # OUTPUT:
    # a dictionary containing:
    # parameters of a beta mixtures: alpha, beta, weights (pi), mu (
        component means)
    # var (component variances), and also
    # loop count: how many loops have this algorithm taken to finish?
    N = len(x)
    if cache_para:
        alpha_lst = cache_para['alpha']
        beta_lst = cache_para['beta']
        pi = cache_para['pi']
    else:
        y1 = np.random.uniform(low=0, high=1, size=1)[0]
        y = np.array([y1])
        while len(y) < m:
            candidates = np.array([i for i in x if i not in y])
            Dy = np.array([np.min(np.abs(y - c)) for c in candidates])**2
            new_y_index = list(np.random.multinomial(1, Dy/sum(Dy)))
            new_y_index = new_y_index.index(max(new_y_index))
            y = np.append(y, candidates[new_y_index])
        y.sort()
        alpha_lst = np.random.rand(m)
        beta_lst = np.random.rand(m)
        pi = np.array([1/m]*m)

        for component_num in range(m):
            samples = np.array([sample for sample in x if (sample <= y[
                component_num]+0.5) and
                                (sample >= y[component_num] - 0.5)])
            # print(samples)
            mu = np.mean(samples)
            if len(samples) <= 1:
                samples = x
            sigma2 = np.std(samples)**2
            phi = mu*(1-mu)/sigma2 - 1
            alpha_lst[component_num] = mu * phi
            beta_lst[component_num] = (1 - mu) * phi

    loop_count = 0
```

```
    start_time = time.perf_counter()
    while True:
        loop_count += 1
        W = np.array([[pi[j] * beta.pdf(x[i], a=alpha_lst[j], b=beta_lst[
            j]) /
                      np.dot(pi, [beta.pdf(x[i], a=alpha_lst[k], b=
                          beta_lst[k])
                                  for k in range(m)])
                      if not np.isnan(pi[j] * beta.pdf(x[i], a=alpha_lst
                          [j], b=beta_lst[j]) /
                      np.dot(pi, [beta.pdf(x[i], a=alpha_lst[k], b=
                          beta_lst[k])
                                  for k in range(m)])) else np.random.
                                      uniform(low=0, high=1, size=1)[0]
                      for j in range(m)] for i in range(N)])

        prev_pi, prev_alpha, prev_beta = pi, alpha_lst, beta_lst
        pi = np.sum(W, axis=0) / N
        mu = np.dot(x, W) / (N * pi)
        sigma2 = np.array([np.dot(W[:, j], (x - mu[j])**2) / (N * pi[j])
            for j in range(m)])
        phi_lst = mu * (1 - mu) / sigma2 - 1
        alpha_lst, beta_lst = mu * phi_lst, (1 - mu) * phi_lst

        my_cond = True
        for i in range(m):
            if not (check_cond(alpha_lst[i], prev_alpha[i], acc)
                    & check_cond(prev_beta[i], beta_lst[i], acc)
                    & check_cond(prev_pi[i], pi[i], acc)):
                my_cond = False
                break
        if my_cond | (time.perf_counter()-start_time >= 5.0 * 3600
            /6/10/1000):
            break

    return {'alpha': alpha_lst, 'beta': beta_lst, 'pi': pi, 'mu': mu, '
        var': sigma2,
            'loop_count': loop_count}


def plot_beta_mixture(alphas, betas, pi, label):
    # plot a beta mixture
    x = np.arange(0.01, 0.99, 0.01)
    y = np.array([0.]*len(x))
    for i in range(len(alphas)):
        y += pi[i] * beta.pdf(x, a=alphas[i], b=betas[i])
    plt.plot(x, y, label=label)


def plot_est_and_true(beta_bandit, arm, para, samples):
    # plot a beta mixture given a bandit arm
    # INPUT:
    # beta_bandit: a beta_bandit instance from the MixtureBanditBeta
```

```
        class
    # arm: which arm?
    # para: dictionary containing alphas, betas, and weights (from EM-M
        algorithm)
    # samples: samples

    # OUTPUT:
    # a plot containing histogram, a theoretical beta mixture densities,
        an estimated
    # beta density using parameters worked out by EM-M algorithm
    ind = arm - 1
    plot_beta_mixture(alphas=beta_bandit.alpha_list[ind],
                      betas=beta_bandit.beta_list[ind],
                      pi=beta_bandit.pi[ind],
                      label='True Distribution')

    plt.hist(samples, bins=50, density=True)

    plot_beta_mixture(alphas=para['alpha'],
                      betas=para['beta'],
                      pi=para['pi'],
                      label='EM Estimated Distribution')
    plt.legend()
    plt.show()


def ucb_v_beta(bandit, n: int, em=False):
    # UCB-V Algorithm for Beta Mixture Bandits
    # bandit: a beta mixture bandit instance
    # n: horizon
    # em: use EM or not?
    # OUTPUT:
    # [[action 1, reward 1],...,[action n, reward n]]
    k = bandit.arm_num
    m = bandit.components_num
    samples = []
    mu_lst = np.array([0.] * k)
    sd_lst = np.array([0.] * k)
    cache_alphas = np.random.rand(k, m)
    cache_betas = np.random.rand(k, m)
    cache_pi = np.random.rand(k, m)
    count_lst = [0] * k
    loop_count = ['init']

    each_arm_sample_size = int(np.min([m * 10, np.floor(n / k)]))
    for i in range(k):
        initial_samples = bandit.pull(arm=i+1, size=each_arm_sample_size)
        for sample in initial_samples:
            samples.append([i+1, sample])
        mu_lst[i] = np.mean(initial_samples)
        sd_lst[i] = np.std(initial_samples)
        count_lst[i] += each_arm_sample_size

```

```
    ucb_lst = 0
    for t in range(k*each_arm_sample_size, n):
        if em:
            if t % 10 == 0:
                print(f"EM-beta, t = {t}, bernstein bounds = {ucb_lst},
                    count = {count_lst},"
                      f" loop count is {loop_count[-1]}")
        else:
            if t % 200 == 0:
                print(f"Non-EM-beta, t = {t}, bernstein bounds = {ucb_lst
                    }, count = {count_lst}")

        ucb_lst = [ucb_bern_bound(t=t, big_t=count_lst[i],
                                  mu=mu_lst[i],
                                  var=sd_lst[i] ** 2,
                                  b=1) for i in range(k)]
        action = ucb_lst.index(max(ucb_lst)) + 1
        new_x = bandit.pull(arm=action, size=1)[0]
        samples.append([action, new_x])
        count_lst[action-1] += 1
        prev_samples = np.array([x[1] for x in samples if x[0] == action
            ])
        mu_lst[action-1] = np.mean(prev_samples)
        if not em:
            sd_lst[action-1] = np.std(prev_samples)
        else:
            if len(prev_samples) <= 50:
                para = em_beta(prev_samples, m, acc=0.1)
            else:
                para = em_beta(prev_samples, m, acc=0.1,
                               cache_para={'alpha': cache_alphas[action
                                   -1],
                                           'beta': cache_betas[action-1],
                                           'pi': cache_pi[action-1]})
            var = np.dot(para['pi'], para['var']+para['mu']**2) - mu_lst[
                action-1]**2
            sd_lst[action - 1] = np.sqrt(var)
            cache_alphas[action-1] = para['alpha']
            cache_betas[action-1] = para['beta']
            cache_pi[action-1] = para['pi']
            loop_count.append(para['loop_count'])
    print(count_lst)
    return samples


def ucb_v_em_beta(bandit, n):
    return ucb_v_beta(bandit, n, em=True)


# sample = my_beta_mixture.pull(arm=1, size=5000)
# print(my_beta_mixture.parameters(latex=True))
#
# para_est = em_beta(sample, m=my_beta_mixture.components_num, acc=0.01)
```

```
# plot_est_and_true(my_beta_mixture, arm=1, para=para_est, samples=sample)


def make_plots_beta(k_lst, components_lst, n_lst, alpha_range_lst, beta_range_lst,
                    rep, folder_name, algorithm_lst, plots_num=1):
    # function for drawing and saving multiple plots and parameters
    # k_lst: a list containing the number of arms an environment carries
    # components_lst: a list containing possible numbers of components
    # n_lst: a list containing horizons
    # alpha_range_list: list of the form [[a1, b1], ..., [an, bn]] denoting the range of alphas
    # beta_range_list: list of the form [[a1, b1], ..., [an, bn]] denoting the range of betas
    # rep: how many experiments to repeat under a given environment
    # folder_name: folder name
    # algorithm_lst: which algorithms to test on? of the form [[algorithm, name],...]
    # plots_num: repeat all for how many plots?
    for a_plot in range(plots_num):
        for k in k_lst:
            for index in range(len(alpha_range_lst)):
                for m in components_lst:
                    print(f"k = {k}, m = {m}, alpha range is {alpha_range_lst[index]}, "
                          f"beta range is {beta_range_lst[index]}".center(130, '@'))
                    my_beta_mixture = MixtureBanditBeta(arm_num=k,
                                                        alpha_range=alpha_range_lst[index],
                                                        beta_range=beta_range_lst[index],
                                                        components_num=m)
                    regret_comparison(bandit=my_beta_mixture,
                                      algorithm_list=algorithm_lst,
                                      n=n_lst[index], rep=rep, show=False, folder_name=folder_name)


make_plots_beta(k_lst=[2],
                alpha_range_lst=[[0, 2], [0, 10]],
                beta_range_lst=[[0, 2], [0, 10]],
                n_lst=[1000, 1000],
                components_lst=[3],
                rep=10,
                folder_name='Beta_mixture_m_3',
                algorithm_lst=[[ucb_v_beta, 'ucb-v-beta'],
                               [ucb_v_em_beta, 'ucb-v-em-beta']])
```

## A.8. regret_comparison.py

```
from Thompson_Sampling import *
from UCB_Normal import *
from UCB_V import *
import os
import pandas as pd
import time
import matplotlib.pyplot as plt
import seaborn as sns


def regret_comparison(bandit, algorithm_list, n, rep, folder_name, show: bool):
    # function to compare regrets and save or display one plot,
    # also saving the parameters to a txt file.

    # bandit: a bandit instance, either gaussian or beta
    # algorithm_list: [[algorithm, algorithm name], ...]
    # n: horizon
    # rep: how many repeated experiments for this bandit problem?
    # folder_name: folder name
    # show: boolean, show the plot immediately or save it?

    print(bandit.parameters(latex=False))
    [optimal_mean, optimal_action] = [bandit.optimal_arm_mean, bandit.optimal_arm]
    optimal_samples = [(i + 1) * optimal_mean for i in range(n)]
    print(f"The optimal action is the arm {optimal_action}. \n\n")

    time_record = []
    regret_df = pd.DataFrame({})
    for [algorithm, algorithm_name] in algorithm_list:
        print(f" {algorithm_name} ".center(120, '#'))
        print(bandit.parameters(latex=False))
        total_time = 0

        for i in range(rep):
            print('{0}, {1:1d}th iteration'.format(algorithm_name, i + 1))

            tic = time.perf_counter()
            samples = algorithm(bandit, n)
            toc = time.perf_counter()
            total_time += toc - tic

            pseudo_means = [bandit.weighted_mean[x[0] - 1] for x in samples]
            regret = np.array(optimal_samples) - np.cumsum(pseudo_means)
            regret = pd.DataFrame({'Cumulative Regret': regret,
                                   'Algorithm / Time': algorithm_name,
                                   't': np.arange(n) + 1})
            regret_df = pd.concat([regret_df, regret], axis=0)
```

```
48
49              print(f"Time: {toc - tic:0.3f} secs, approximately remaining:
                    "
50                    f" {total_time / (i + 1) * (rep - i - 1) / 60:0.2f}
                          minutes. \n")
51
52          time_record.append(total_time / rep)
53
54      for i in range(len(time_record)):
55          print(f"{algorithm_list[i][1]} on average {time_record[i]:0.3f}
                secs per experiment.")
56          name_with_time = f"{algorithm_list[i][1]} / {time_record[i]:0.3f}
                "
57          regret_df.loc[regret_df['Algorithm / Time'] == algorithm_list[i
                ][1],
58                        'Algorithm / Time'] = name_with_time
59
60      if bandit.is_beta:
61          [alpha1, alpha2] = bandit.alpha_range
62          [beta1, beta2] = bandit.beta_range
63          name_info = f"k = {bandit.arm_num:d},  m = {bandit.components_num
                :d},  n = {n:d},  r = {rep:d}"
64          name_identify = f"{name_info}, [{alpha1}, {alpha2}], [{beta1}, {
                beta2}]"
65          name_without_suffix = name_identify
66      else:
67          [mean1, mean2] = bandit.mean_range
68          [sd1, sd2] = bandit.sd_range
69          name_info = f"k = {bandit.arm_num:d},  m = {bandit.components_num
                :d},  n = {n:d},  r = {rep:d}"
70          name_identify = f"{name_info},  [{mean1}, {mean2}]"
71          name_without_suffix = name_identify
72
73      print('\n\nSaving parameters......\n\n')
74
75      try:
76          os.mkdir(os.path.join(os.getcwd(), folder_name))
77          os.mkdir(os.path.join(os.getcwd(), folder_name, 'parameters'))
78          os.mkdir(os.path.join(os.getcwd(), folder_name, 'plots'))
79      except FileExistsError:
80          pass
81
82      my_int = 1
83      while os.path.exists(os.path.join(os.getcwd(), folder_name, '
            parameters', name_identify)):
84          name_identify = f"{name_without_suffix} ({my_int})"
85          my_int += 1
86
87      my_directory_para = os.path.join(os.getcwd(), folder_name, '
            parameters', name_identify)
88      my_directory_png = os.path.join(os.getcwd(), folder_name, 'plots',
            name_identify)
89      f = open(my_directory_para, 'w+')
```

```
    f.write(bandit.parameters(latex=True))
    f.close()

    print('Saving plots......\n\n')
    sns.set(style="whitegrid", palette="muted", color_codes=True)
    sns.lineplot(data=regret_df, x="t", y='Cumulative Regret', hue='Algorithm / Time')
    plt.rcParams.update({'font.size': 100})

    if bandit.is_beta:
        plt.title(f"{name_info},  ${alpha1} < a < {alpha2}$,  ${beta1} < b < {beta2}$",
                  fontsize=14)
    else:
        plt.title(f"{name_info},  ${mean1} < \mu < {mean2}$,  ${sd1} < \sigma < {sd2}$",
                  fontsize=14)
    plt.savefig(f"{my_directory_png}.png")
    print(' DONE! '.center(120, '#')+'\n\n')
    if show:
        plt.show()
    plt.clf()


def make_plots(k_lst, mean_range_lst, n_lst, components_num, rep, folder_name, algorithm_lst, plots_num=1):
    # function for drawing and saving multiple plots and parameters

    # k_lst: a list containing the number of arms an environment carries
    # components_lst: a list containing possible numbers of components
    # n_lst: a list containing horizons
    # mean_range_list: list of the form [[a1, b1], ..., [an, bn]] denoting the range of means
    # rep: how many experiments to repeat under a given environment
    # folder_name: folder name
    # algorithm_lst: which algorithms to test on? list of the form [[algorithm, name], ...]
    # plots_num: repeat all for how many plots?

    for a_plot in range(plots_num):
        for k in k_lst:
            for index in range(len(mean_range_lst)):
                for m in components_num:
                    print(f"k = {k}, m = {m}, mean range is {mean_range_lst[index]}".center(130, '@'))
                    bandit_1 = MixtureBandit(arm_num=k,
                                             components_num=m,
                                             mean_range=mean_range_lst[index],
                                             sd_range=[0.5, 1])
                    regret_comparison(bandit_1,
                                      n=n_lst[index],
                                      rep=rep,
```

```
                                        folder_name=folder_name,
                                        show=False,
                                        algorithm_list=algorithm_lst)


def algo_lst(index):
    algo = [[ucb_v, 'UCB-V'],                                   # 1
            [ucb_v_em, 'UCB-V-EM'],                             # 2
            [ucb_normal, 'UCB1-Normal'],                        # 3
            [ucb_normal_em, 'UCB1-Normal-EM'],                  # 4
            [thompson_sampling, 'Thompson Sampling'],           # 5
            [ucb, 'UCB'],                                       # 6
            [ucb_em, 'UCB-EM']]                                 # 7
    return [algo[i-1] for i in index]


make_plots(k_lst=[2, 4, 8],
           mean_range_lst=[[0, 1], [0, 5]],
           n_lst=[1000, 500],
           components_num=[2, 3, 4],
           rep=20,
           folder_name='Thompson_sampling',
           algorithm_lst=algo_lst([2, 5]))
```