How kernelized Multi-Armed Bandit algorithms compare to other algorithms with fixed kernelized reward and noisy observations

Marijn Herrebout

Supervisor(s): Julia Olkhovskaia

EEMCS, Delft University of Technology, The Netherlands

Abstract

The aim of this paper is to challenge and compare several Multi-Armed Bandit algorithms in an environment with fixed kernelized reward and noisy observations. Bandit algorithms are a class of decision-making problems with the goal of optimizing the trade-off between exploration and exploitation of all choices. Each decision yields some reward, and the goal is to minimize the regret that follows from a combination of decisions, that is to say to minimize the difference between the set of decisions made, and the set of optimal decisions. In particular, these algorithms deal with the trade-off between choosing the best-known option and exploring new, possibly better options. These algorithms are widely used in reinforcement learning, optimization and economics, where decisions need to be made without all the information and with some uncertainty. Each environment is different however, and some algorithms are better in some environments than others.

1 Introduction

The class of Multi-Armed Bandits (henceforth referred to as MAB) problems is large, including even problems that don't appear like they belong. They are used for A/B testing, advertisement placement on a webpage, network routing, dynamic pricing, waiting problems, resource allocation, tree search, and many more different types of problems [1]. Most of these problems are important for financial reasons, either by minimizing cost or maximizing income, and others are about convenience. While the general concept of MABs can be applied to all these problems, they have different environments, different algorithms, and different ways of solving them. That is also the area in which this paper contributes to the literature; by focusing on a select few algorithms and environments, we can identify the (near-)optimal solution for this setting. In this paper we look at four specific algorithms and compare them in runtime, memory consumption, and regret. We will formalize the notion of regret in a later chapter.

The problems we look at in this paper are stochastic bandits (Upper Confidence Bound, UCB [1][2]), adverserial bandits (Exp3 [3][2]), linear contextual bandits (LinUCB [4][2]), and contextual kernelized bandits (KernelUCB [5][2]). We explore the influence the environment has on these different algorithms in our final comparison of each algorithm, and also explore different kernel functions for use in the KernelUCB algorithm to find which function fits best with our environment.

In Chapter 2 the basic goal and methodology of MABs is introduced, and Chapter 3 dives into each of the mathematical algorithms and functions that are used. Chapter 4 shows the resuls of the comparisons. With a brief intermission on responsible research and ethics in Chapter 5, the results and future work is discussed in Chapter 6, to finally conclude the paper in Chapter 7.

2 Methodology

To provide a fair comparison between the UCB, Exp3, LinUCB and GP-UCB algorithms, it is important that we evaluate them in the same setting with the same criteria. To that end, we compare them using SMPyBandits, an open-source Python library that provides a good structure through which to implement, test, and compare different MAB problems.

One of the most important benchmarks with which we will compare the algorithms is with regret relative to the optimal solution. The definition of regret in the context of MABs is very similar to its definition in the English language. The measure of regret that a particular algorithm incurs is the difference of the reward achieved by that algorithm minus the reward achieved by some optimal algorithm. In other words, the value represents the regret for choosing the sub-optimal algorithm over the optimal algorithm. This can be formalized mathematically as shown in equation 1 below, where R_n is the regret of the algorithm, n is the number of trials, $\mathcal{A}$ is the set of all actions, and X_t is the reward observed at time t [1].

$$R_n = n \max_{a \in \mathcal{A}} \mu_a - \mathbb{E}\left[\sum_{t=1}^{n} X_t\right] \tag{1}$$

To put this into English, the optimal reward is found by taking the highest mean μ_a of all the actions n times. The difference between this optimal reward and the actually observed reward then becomes R_n, which we aim to minimize by finding X_t as close as possible to μ_a where a is optimal. This definition of regret can be applied to every MAB algorithm, and is therefore a good heuristic for comparing them.

3 Algorithms

The algorithm to use in a problem setting where an MAB may be applied depends on the nature of the problem. Some algorithms are tailored to certain environments and as a result perform better in similar situations, whereas they may perform much worse in other settings. Choosing the right algorithm for a problem is important, and as such we need to get to know the algorithms first. In this Chapter we will briefly go over each algorithm, explain its foundation and how it chooses arms.

3.1 Upper Confidence Bound

The Upper Confidence Bound algorithm, UCB, is a rather simple algorithm that achieves sublinear regret. Based on the sampled reward of an arm, it assigns that arm a certain upper confidence bound value, which will in all likelihood be an overestimate of the actual mean of the arm. Upon choosing an arm, the observed mean of the arm is updated and the number of times the arm has been chosen is incremented by one. These variables determine the weight of the upper confidence bound. Equation 2 describes this process mathematically.

$$A_t = argmax_i \left(\hat{\mu}_i(t-1) + \sqrt{\frac{2\log(t)}{T_i(t-1)}}\right) \tag{2}$$

In this equation A_t is the arm that is chosen at time t by testing the confidence bound for all arms and choosing the one with the highest result. In the inner function, $\hat{\mu}_i(t-1)$ is the total observed mean as of time $t-1$ for arm i, and $T_i(t-1)$ is the number of times that we have visited arm i since $t-1$. Other implementations of the UCB algorithm exist that include a confidence level variable δ, which allows for more fine-tuned control of how quickly the upper confidence bound scales with time. Choosing a good value for δ is an optimization problem in its own right, as choosing it too low will lead to excessive exploration of sub-optimal arms, whereas choosing it too high will not lead to the optimal arm. The algorithm that was used in this paper is the one described in Equation 2.

3.2 Exp3

Adversarial bandits, the bandits we consider with the Exp3 algorithm, are similar to stochastic bandits, but with some key differences [1][3]. For one, the reward function changes over time by a certain weight. Furthermore, the way the arms are chosen is drastically different from UCB. What makes these bandits adversarial as opposed to other types of bandits, is that these model a scenario that acts as though there is an adversary influencing the rewards.

$$P_{ti} = \frac{exp(\gamma \hat{S}_{t-1,i})}{\sum_{j=1}^{k} exp(\gamma \hat{S}_{t-1,j})} \tag{3}$$

$$\hat{S}_{ti} = \hat{S}_{t-1,i} + 1 - \frac{\mathbb{I}\{A_t = i\}(1 - X_t)}{P_{ti}} \tag{4}$$

$$A_t \sim P_t$$

Equations 3 and 4 form the basis of the Exp3 algorithm using exponential weighting [1][6]. P_{ti} is the sampling distribution calculated at each time step, $\hat{S}_{ti}$ is the cumulative reward for arm i at time t, and γ is the learning rate of the algorithm. If γ is small, the algorithm will do a lot of exploration and relatively little exploitation, whereas higher values lead to more exploitation than exploration.

3.3 LinUCB

The next class of bandits are contextual ones, where each arm also has a vector associated with it known as the context of that arm [4]. The features that make up the context vectors are used to estimate the upper confidence bound in LinUCB. This is similar to UCB, but with context features rather than means.

$$p_{t,a} = \theta_t^\top x_{t,a} + \alpha\sqrt{x_{t,a}^\top M_t^{-1} x_{t,a}} \tag{5}$$

$$A_t = argmax_a \, p_{t,a}$$

$$M_{t+1} = M_t + x_{t,a_t} x_{t,a_t}^\top$$

$$b_{t+1} = b_t + x_{t,a_t} r_t$$

$$\theta_t = M_t^{-1} b_t$$

In LinUCB shown in Equation 5, we have a weights vector θ that determines how much a context feature influences the upper confidence bound, and α is the learning rate. The algorithm starts with M equal to the identity matrix, and is updated to become $M = M + x_{t,a_t} x_{t,a_t}^\top$ at the end of each time step, along with $b = b + x_{t,a_t} r_t$, where r_t is the reward obtained at time t. These two variables A and b are then used at the beginning of each round to update the weights vector: $\theta_t = A^{-1} b$.

3.4 KernelUCB

Kernelized bandit problems are part of a class of (possibly) non-linear problems that algorithms like UCB, Exp3, and even LinUCB may not provide an adequate solution to. A good explanation of kernelized problems was given in the paper introducing the specific algorithm used here:

"There are many situations in which an environment repeatedly provides an agent with a very large number of actions together with some contextual information (Cesa-Bianchi & Lugosi, 2006). These actions yield rewards when chosen and the agent wants to continually choose actions that yield high expected reward while not having enough time to explore them all. Thus it is natural to learn a relationship between the context provided for each action and the expected reward it produces. Kernel methods (Shawe-Taylor & Cristianini, 2004) provide a way to extract from observations possibly non-linear relationships between the contexts and the rewards while only using similarity information between contexts." [5, p. 1]

$$\sigma_{t,a} = \sqrt{k(x_{t,a}, x_{t,a}) - k_{x,t}^\top K_t^{-1} k_{x,t}}$$

$$u_{t,a} = k_{x,t}^\top K_t^{-1} y_t + \frac{\eta}{\sqrt{\gamma}} \sigma_{t,a} \tag{6}$$

$$A_t = argmax_a u_{t,a}$$

Equation 6 offers a brief overview of the variables calculated directly for each arm, where k is the kernel function, K^{-1} the kernel matrix inverse, η the exploration rate, and γ the learning rate. A more detailed explanation of the precise workings of the KernelUCB algorithm can be found in the 2022 paper that introduced it [5].

Kernel functions

Kernel functions are an invaluable tool for extracting patterns and relations in non-linear settings. In machine learning methods, decision boundaries are often used to distinguish different classes based on their features. These boundaries are often linear, but these do not provide optimal results for non-linear problems [7]. This area is where kernel functions excel; they are linear classifiers for non-linear problems. Most kernel functions described in this chapter have a parameter l which represents the length scale. This can be either a scalar, or a vector with the same number of dimensions as the inputs [8]. The kernels covered in this Chapter work best in distinct environments, sometimes performing worse than the rest and other times better.

Radial Basis Function

The Radial Basis Function kernel, as shown in Equation 7, is also known as the square exponential kernel, as it uses the (Euclidean) distance squared between the two inputs. This kernel is also known as RBF [8].

$$k(x_i, x_j) = exp\left(-\frac{d(x_i, x_j)^2}{2l^2}\right) \tag{7}$$

Rational Quadratic

The Rational Quadratic kernel in Equation 8 is similar to RBF, the Radial Basis Function in that it can be seen as an infinite sum of RBF kernels with different length scales [9][10].

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha} \tag{8}$$

Matern

The Matern kernel function in Equation 9 is a generalization of RBF with an extra parameter v that determines the smoothness of the function, with the function being less smooth as v gets smaller; as v increases, Matern will, in the limit, be equivalent to RBF. In this equation, K_v is a modified Bessel function, and Γ is the gamma function [11][10][12].

$$k(x_i, x_j) = \frac{K_v}{\Gamma(v)2^{v-1}} \left(\frac{\sqrt{2v}}{l} d(x_i, x_j)\right)^v \left(\frac{\sqrt{2v}}{l} d(x_i, x_j)\right) \tag{9}$$

ExpSineSquared

The ExpSineSquared kernel in Equation 10 is able to model functions that repeat themselves exactly. In this equation, p is the periodicity of the kernel [13].

$$k(x_i, x_j) = exp\left(-\frac{2sin^2(\pi d(x_i, x_j)/p)}{l^2}\right) \tag{10}$$

4 Results

In order to establish how the results that are about to be introduced were established, it is important to discuss the hyper-parameters and environments first. All the results below were produced using three 20-dimensional arms, with fixed arbitrary context vectors over 10 repetitions. These vectors are shown below.

$x_1 = [0.3, 0.5, 0.2, 0.4, 0.6, 0.3, 0.5, 0.2, 0.4, 0.6, 0.3, 0.5, 0.2, 0.4, 0.6, 0.3, 0.5, 0.2, 0.4, 0.6]$
$x_2 = [0.7, 0.3, 0.3, 0.2, 0.6, 0.7, 0.3, 0.3, 0.2, 0.6, 0.7, 0.3, 0.3, 0.2, 0.6, 0.7, 0.3, 0.3, 0.2, 0.6]$
$x_3 = [0.4, 0.6, 0.1, 0.2, 0.7, 0.4, 0.6, 0.1, 0.2, 0.7, 0.4, 0.6, 0.1, 0.2, 0.7, 0.4, 0.6, 0.1, 0.2, 0.7]$

Furthermore, these contexts have covariance matrices equivalent to a 20-dimensional identity matrix multiplied by 0.5. The arms also have noise to simulate noisy observations. This noise follows a normal distribution with a mean of 0 and variance of 0.01.

The reward functions that were tested take an input x which is defined as the dot product between the context of the arm and a context weights vector. This is shown in Equation 11. x, the input of the reward function, is not to be confused with $\overrightarrow{x_i}$, which is the context of arm i. To make this distinction clearer, the latter is shown with an overhead arrow to explicitly indicate it is a vector. The context weight vector $\overrightarrow{\theta^*}$ is a 20-dimensional vector filled with 0.5.

$$x = \overrightarrow{\theta^*} \cdot \overrightarrow{x_i} \tag{11}$$

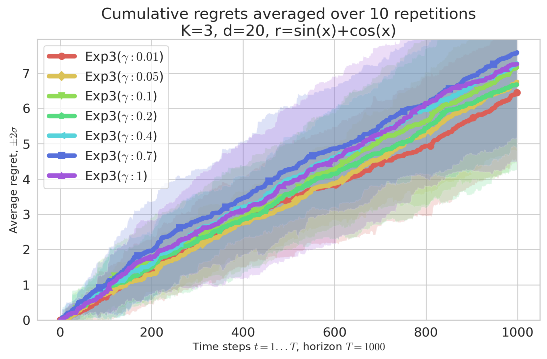

Figure 1: *Cumulative regret with different γ values with Exp3*

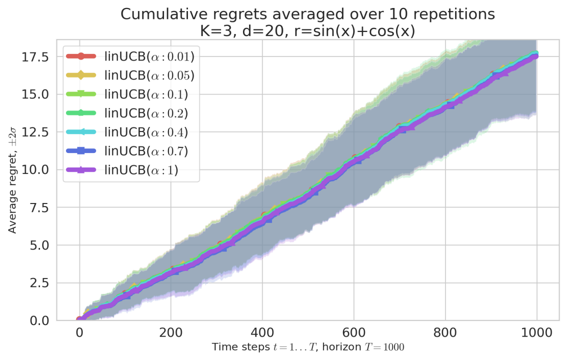

Figure 2: *Cumulative regret with different α values with LinUCB*

4.1 Hyper-parameters

As Exp3, LinUCB, and the kernel functions used in KernelUCB all have hyper-parameters, these were all individually plotted with different values. These hyper-parameters were tuned using the reward function $r = sin(x) + cos(x)$. A silent extra term ϵ in this and all future reward function could be imagined to represent the noise value drawn from a normal distribution with a mean of 0 and variance of 0.01, but this is omitted for simplicity. The difference the reward function makes on the cumulative regret will be explored later in this Chapter.

Figure 1 shows the cumulative regret of the Exp3 algorithm with different γ learning rate values, ranging from 0.01 to 1.0. The figure shows that the value of γ does not have any significant influence on the cumulative regret. The best performing algorithm has a value of $\gamma = 0.01$, so that value will be used to represent Exp3 in future comparisons.

For the LinUCB algorithm we have a learning rate hyper-parameter α, but as seen in Figure 2, this has negligible impact on the performance (cumulative regret) of the algorithm. Nonetheless, a value of $\alpha = 0.01$ will be used in future algorithm comparisons.

The KernelUCB algorithm could be seen as having the kernel itself being a hyper-parameter. However, as the results later in this Chapter will show, the kernel plays a major role in the performance of the algorithm, depending on the reward function among other things. Since the kernel function

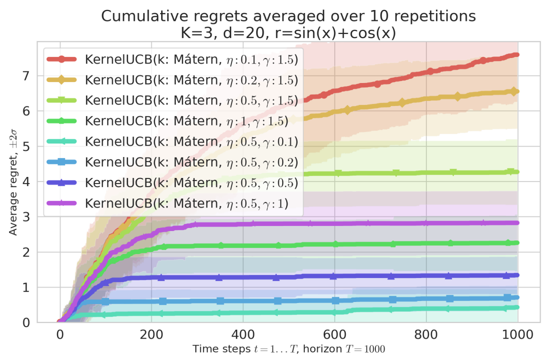

Figure 3: *Cumulative regret with different η and γ values with KernelUCB with the Mátern kernel*

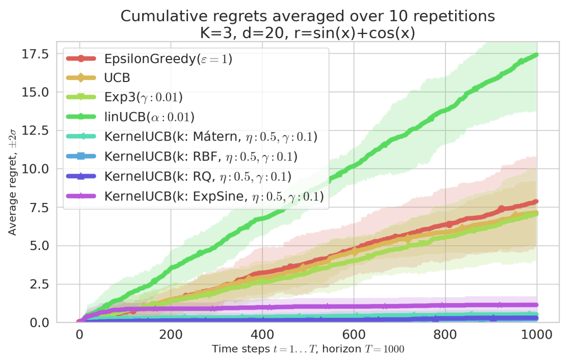

Figure 4: *Cumulative regret with $r = sin(x) + cos(x)$*

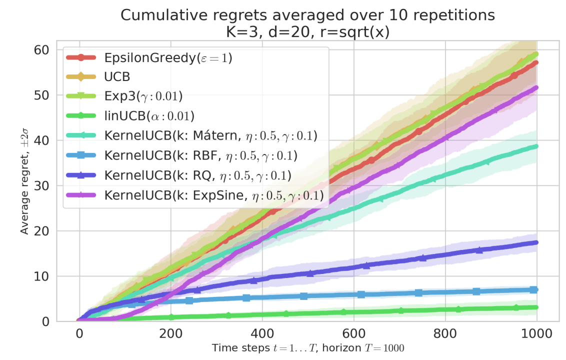

Figure 5: *Cumulative regret with $r = \sqrt{x}$*

changes the performance, different kernels are treated as different algorithms entirely. Therefore, all comparisons will list all kernels considered in this paper.

KernelUCB does have two different hyper-parameters though, the exploration rate η and the learning rate γ. A crude comparison in cumulative regret, or simply regret, is shown in Figure 3. These hyper-parameters were arbitrarily compared with the Mátern kernel, as the kernel itself does not influence the hyper-parameters. It is easy to see from this figure that these hyper-parameters have a significant impact on the regret of the algorithm, with the best combination being $\eta = 0.5, \gamma = 0.1$. These are also used for future comparisons in this paper.

4.2 Reward functions

Having established good hyper-parameter values for all the algorithms to be considered, the next variable in the performance of the algorithms is the reward function. Together with the contexts and the arms, these form the environments that can model concrete problems. We specifically turn our attention to non-linear reward functions, as that is the domain in which kernelized methods traditionally excel over linear algorithms like UCB and Exp3.

UCB and Exp3 used the observed means as the deciding factor in what arm to choose, but this turns out to be a bad decision in non-linear settings where the context plays an important role. This is one possible explanation as to why you will never see these two algorithms converge, and why they will never be the best algorithm to use in the settings we explore.

A new algorithm ϵ-Greedy that has not been mentioned before is introduced here with a γ exploration value of 1.0. This means the algorithm will make a random decision with probably 1.0. This algorithm displays the regret we would achieve if we made a completely random decision.

Figure 4 shows the difference in regret for the reward function $r = sin(x) + cos(x)$. This figure immediately stands out in that all applications of KernelUCB converge, and that they are all very close to optimal regret. On the other hand, random choice, UCB, Exp3 and LinUCB all struggle to converge with this periodic reward function. This is easy to explain; the linear algorithms are intended to be used for linear reward functions that grow linearly with time. However, with this particular function, regret changes over time due to the nature of sin and cos. During the peak of this reward function, these algorithms think this is a good choice, but once it goes negative it incurs a large amount of regret. This repeats with every period of the reward function, and is therefore never able to converge.

With a reward function $r = \sqrt{x}$, we notice a different trend. In Figure 5 we can still see the linear algorithms fail to converge, but we also notice a difference between different kernel functions in KernelUCB. In particular, ExpSine and Mátern turn out not to be good fits for this reward function as they are only slightly better than random, UCB, and Exp3. RQ still does well, but it does not converge as well as RBF, which is the best kernel function for this environment using KernelUCB. The winner is LinUCB though, which is a curious result given the reward function is non-linear. It is not clear how it performs so well, but KernelUCB with RBF is in a close second place. One possible explanation is that LinUCB is still a good option in non-linear environments as long as they are not periodic, but this is put to the test in other environments.

If we increase the horizon from 1000 to 2000 and focus on RBF and LinUCB, we can get a better picture as to how these two algorithms compare. As Figure 6 shows, RBF takes longer to yield low regret, whereas LinUCB is consistent from the beginning in its regret. Neither of them actually converge, but from around $t = 500$ onward, they increase

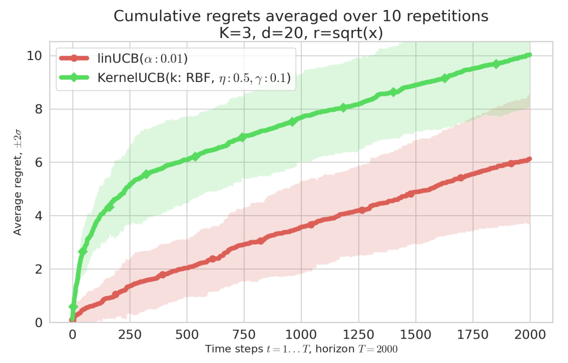

Figure 6: *Cumulative regret with* $r = \sqrt{x}$ *for* $T = 2000$

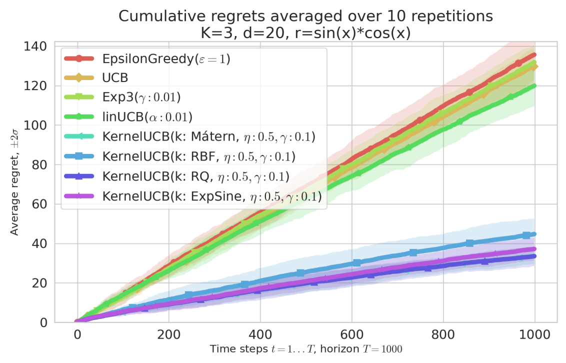

Figure 9: *Cumulative regret with* $r = sin(x)cos(x)$

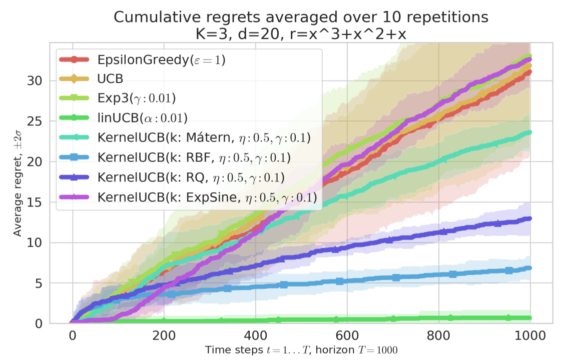

Figure 7: *Cumulative regret with* $r = x^3 + x^2 + x$

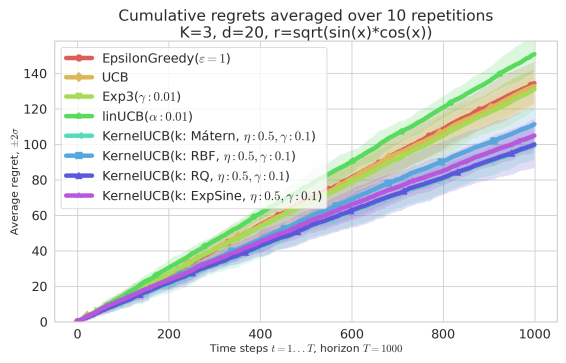

Figure 10: *Cumulative regret with* $r = \sqrt{sin(x)cos(x)}$

quite similarly in their regret, which means their difference in performance comes mainly from the first few hundred iterations.

With the polynomial reward function $r = x^3 + x^2 + x$ we notice a lot of variance in Figure 7. It also shows that ExpSine is one of the worst options to choose, with it performing even worse than pure random choice. RBF is still a decent option, but LinUCB outperforms all of the other algorithms.

Figure 8 shows the strength of KernelUCB clearer than any other plot with reward function $r = \frac{sin(x)}{x^2}$. All kernel functions converge and yield very low regret, whereas random,

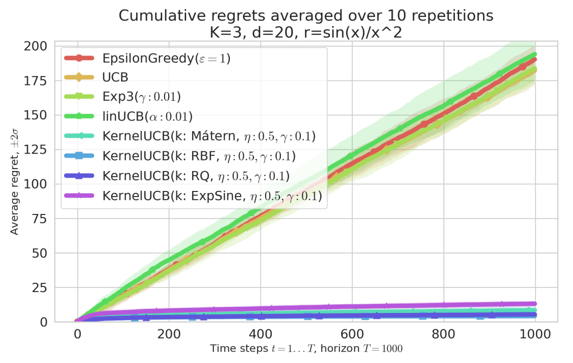

Figure 8: *Cumulative regret with* $r = \frac{sin(x)}{x^2}$

UCB, Exp3, and even LinUCB fail to converge and incur a lot of regret. This plot also supports the hypothesis that LinUCB is unable to converge in environments with a period to them, being the $sin(x)$ component of the reward function.

Figure 9 is once again a non-linear period reward function $r = sin(x)cos(x)$. While none of the algorithms converge, KernelUCB does provide far better solutions than the alternatives.

A different reward function $r = \sqrt{sin(x)cos(x)}$ produces interesting results in Figure 10. Although the reward function is non-linear and periodic, none of the algorithms are a good solution to this problem. None of the algorithms converge, and they all have high regret. Despite the similarity with the reward function $r = sin(x)cos(x)$ that is shown in Figure 9, this figure looks quite different. One might expect RBF to do better having seen its performance in Figures 5 and 9, but we can evidently not draw any links between different environments.

Finally, we consider $r = ln(x)$ in Figure 11. This too is a good example of the quality of KernelUCB, but it also shows that LinUCB can be a good choice for some non-linear functions.

4.3 Runtime and Memory Consumption

Although some algorithms are faster than others, the time it takes for them to run and the amount of memory they use up can also be important factors, especially in situations where these resources can be limited such as in IoT devices, or if

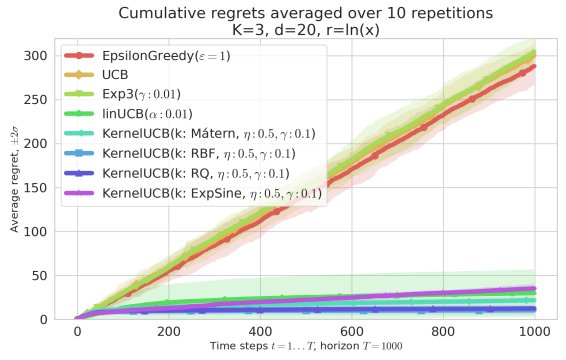

Figure 11: *Cumulative regret with* $r = ln(x)$

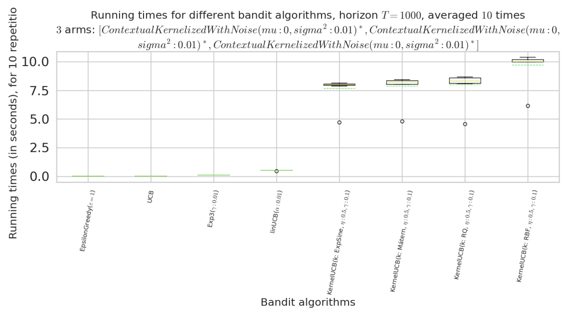

Figure 12: *Runtime in seconds of the algorithms*

results need to be achieved quickly and sub-optimal performance is acceptable.

Figure 12 illustrates quite a significant difference in how fast each algorithm runs. The random choice and algorithms tailored to linear problems like UCB, Exp3 and LinUCB take significantly less time to run than KernelUCB, but we can also see that some kernel functions like RBF are computationally more expensive than others.

Although memory consumption is closely coupled with the details of implementation, Figure 13 unmistakenly shows that KernelUCB uses a lot more memory than the rest of the algorithms.

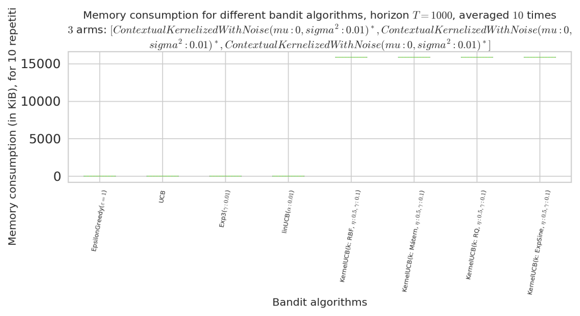

Figure 13: *Memory consumption in kilobytes of the algorithms*

5 Responsible Research

Multi-armed bandit problems are used in a wide variety of fields as listed in the introduction. In particular, the example of A/B testing is a good problem to explore the ethics of MABs further. In A/B testing, the question is asked whether to decide on option A or on option B. This could be financially motivated, like which advertisement is better to show a user, but A/B testing is also used (though not widely) in clinical trials to determine treatment plans, and could even be used to determine whether a patient gets the real drug or the placebo [1][14]. It is needless to say that choosing the wrong algorithm for this type of problem has a more substantial impact than if one were to choose the wrong algorithm for an advertisement problem.

Beyond the use of MABs in clinical trials, where the ethical considerations speak for themselves, even in situations where finance plays a role MABs need to be reliable. If they are used for dynamic pricing in an automated system, the retailer can incur significant losses if the price goes too low for erroneous reasons, or lose out on income if the price skyrockets unprompted.

The results produced for this paper will not be perfectly reproducible, as there is a randomization aspect to them. As the algorithms do not use real-world data but artificial distributions, it is impossible to achieve the exact same results. Furthermore the running time and memory consumption results depend on the implementation of the algorithms and the architecture that runs them.

6 Discussion and Future Work

The results produced for this paper were made with various reward functions, but the number of arms was fixed, the context vectors were fixed, as well as the number of dimensions and the context weight vector. In real world settings, the odds that these all match up to the values represented in this paper are slim. More extensive research could be done to understand the influence the number of dimensions has on the performance of the algorithms, and more research on the impact the number of arms has would also be greatly beneficial.

On top of that, the hyper-parameters were tuned rather generally on just one single environment. As there are a large number of variables, it is impractical to optimize these all for our very specific scenarios. If these comparisons are done again with a concrete, real-world example, it would be beneficial to give more attention, through grid search or other thorough methods to optimizing the hyper-parameters of the algorithms that are to be used.

Furthermore, if a real-world example is considered, the goal presumably is still to minimize regret. This paper only looks at a small selection from a very large pool of algorithms that may be suitable for the job. Extending our vision to other contextual MAB algorithms, or other kernel functions for KernelUCB, may prove that even better algorithms exist for a specific problem than the ones covered. One such algorithm could be CGP-UCB, for instance [15].

As we briefly covered already, runtime and memory consumption could also play an important role in the application of MABs. Optimizing these algorithms to be faster through vector operations or use less memory may be another challenging yet interesting topic of research.

Lastly, doing a deeper dive into one of the reward functions covered with unexpected results may also yield a better understanding of the algorithms. LinUCB converging faster with less regret than KernelUCB on a non-linear problem may speak to other strengths LinUCB has besides linear problems.

7 Conclusions

In conclusion, it is impossible to crown one particular algorithm as the undeniable winner over any other. We have looked at the definitions of UCB, Exp3, LinUCB and KernelUCB, and looked at several kernel functions for use with KernelUCB. Having tuned the hyper-parameters for each of the algorithms that have them, we compared the algorithms with a wide variety of non-linear reward functions. A few patterns emerged, including some unexpected like LinUCB doing well on some non-linear, non-periodic reward functions. Another rather common occurrence is KernelUCB doing very well with period reward functions, but not every single one that was tested. Having looked at only a few environments, there is plenty of future work that could be done, especially with real-world data. In the end though, we have shown that KernelUCB is a good choice for a lot of non-linear reward functions, especially if they have a periodic element.

Acknowledgements

I would like to extend my sincere thanks to Lilian Besson for creating and open-sourcing SMPyBandits, without which a lot more time a research would be spent creating a structure and framework in which to create Multi-Armed Bandits and implement the relevant algorithms [16]. I would like to thank supervisor Julia Olkhovskaia for the project and her assistance. I would be remiss if I did not also thank my colleagues and teammates Cody Boon, Rafal Owczarski, Dragos Arsene, and Weicheng Hu for their wisdom, shared code, and general support throughout the project.

References

[1] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

[2] D. Arsene, C. M. Boon, M. Herrebout, W. Hu, and R. Owczarski, "Contextual SMPyBandits," Available at https://gitHub.com/thatCbean/SMPyBandits, 2024. [Online]. Available: https://github.com/thatCbean/SMPyBandits/

[3] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The nonstochastic multiarmed bandit problem," *SIAM journal on computing*, vol. 32, no. 1, pp. 48–77, 2002.

[4] W. Chu, L. Li, L. Reyzin, and R. Schapire, "Contextual bandits with linear payoff functions," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop and Conference Proceedings, 2011, pp. 208–214.

[5] M. Valko, N. Korda, R. Munos, I. Flaounas, and N. Cristianini, "Finite-time analysis of kernelised contextual bandits," in *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'13. Arlington, Virginia, USA: AUAI Press, 2013, p. 654–663.

[6] S. Bubeck and N. Cesa-Bianchi, "Regret analysis of stochastic and nonstochastic multi-armed bandit problems," in *Foundations and Trends® in Machine Learning*, January 2012, pp. 1–122.

[7] T. Afonja, "Kernel functions," Available at https://towardsdatascience.com/kernel-function-6f1d2be6091 (11/06/2024).

[8] Scikit Learn, "RBF," Available at https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html (11/06/2024).

[9] Scikit Learn, "RationalQuadratic," Available at https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RationalQuadratic.html (11/06/2024).

[10] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*, ser. Applied mathematics series. Dover Publications, 1965.

[11] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, 2005. [Online]. Available: https://doi.org/10.7551/mitpress/3206.001.0001

[12] Scikit Learn, "Matern," Available at https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.Matern.html (11/06/2024).

[13] Scikit Learn, "ExpSineSquared," Available at https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.ExpSineSquared.html (11/06/2024).

[14] W. H. Press, "Bandit solutions provide unified ethical models for randomized clinical trials and comparative effectiveness research," *Proceedings of the National Academy of Sciences*, vol. 106, no. 52, pp. 22 387–22 392, 2009. [Online]. Available: https://www.pnas.org/doi/abs/10.1073/pnas.0912378106

[15] A. Krause and C. Ong, "Contextual gaussian process bandit optimization," in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds., vol. 24. Curran Associates, Inc., 2011.

[16] SMPyBandits, "SMPyBandits," Available at https://github.com/SMPyBandits/SMPyBandits.