MABWiser: A Parallelizable Contextual Multi-Armed Bandit Library for Python

Emily Strong
Fidelity Investments
Boston, USA
emily.strong@fmr.com

Bernard Kleynhans
Fidelity Investments
Boston, USA
bernard.kleynhans@fmr.com

Serdar Kadıoğlu
Fidelity Investments
Boston, USA
serdar.kadioglu@fmr.com

***Abstract*—Contextual multi-armed bandit algorithms serve as an effective technique to address online sequential decision-making problems. Despite their popularity, when it comes to off-the-shelf tools the library support remains limited, in particular for the Python technology stack. To fill this gap, in this paper we present a system that provides context-free, parametric and non-parametric contextual multi-armed bandit models. The available bandit policies accommodate both batch and online learning. The MABWISER system is implemented as an open-source Python library. Our design enables built-in parallelization to speed up training and test components for scalability while ensuring the reproducibility of results. We present a running example to highlight the user-friendly nature of the public interface and discuss the simulation capability of the library for hyper-parameter tuning and rapid experimentation.**

***Index Terms*—contextual multi-armed bandits, sequential decision making, Python machine learning libraries**

I. Introduction

We frequently face situations in daily life in which we must choose between a known, familiar entity and a lesser known, unfamiliar one. For example, when ordering from the menu at a restaurant, do we go with our favorite dish or do we try out the chef's special? When getting a haircut, do we get a trim or ask for a new style? For these and many similar problems, the choice between different options ("*arms*") depend on our past experience ("*learning*") as well as the particular situation ("*context*") that we are currently in.

Such problems can be addressed using multi-armed bandit (MAB) algorithms [1]. The multi-armed bandit problem comes from the scenario of a gambler choosing which machine to play from a row of slot machines called one-armed bandits with unknown average payouts ("*rewards*"). On each turn the gambler faces a dilemma: choose an arm with the highest average reward so far and receive an expected result ("*exploit*") vs. choose an arm with not much information yet and potentially learn something new ("*explore*"). Overall, the objective is to balance this exploration-exploitation trade-off to maximize the long-term cumulative reward.

In many situations, the state in which the decision is made affects the outcome. These scenarios are known as contextual multi-armed bandits (CMABs). In the restaurant example, our decision might be influenced by the price, ingredients, and our current mood. CMABs can be used to frame many recommendation and optimization problems such as personalized recommendations to maximize click-through rates [2], advertisement optimization to maximize revenue [3], and patient allocation in clinical trials to improve outcomes [4]. With this wide range of problem domains in mind, we designed and developed MABWISER[1]: a Python library for rapid prototyping and experimentation with CMAB algorithms.

TABLE I
Example Data for Advertisement Optimization

Ad	Revenue	Age	ClickRate	Subscriber
1	10	29	0.10	0
2	17	52	0.43	1
3	22	47	0.18	0
4	9	36	0.25	1
1	20	61	0.07	0

Our contributions in this line of research are as follows: We designed a system and released an open-source library that can support context-free, parametric and non-parametric contextual models while enabling offline batch training and online learning for all policies (Section §4). For ease of use, the library follows a *scikit-learn* [5] style public interface (Section §6). The built-in parallelization strategy allows speedups in both training and testing for scalability. We show how to ensure reproducibility between parallel runs, a highly desired property in academic and industrial settings (Section §7). The library includes a simulation capability for hyper-parameter tuning and rapid experimentation (Section §8). Overall, MABWISER can serve the greater AI community for further research and development on multi-armed bandit algorithms.

II. Running Example: Advertisement Optimization

To introduce the idea behind multi-armed bandits and illustrate how MABWISER works, let us consider the example given in Table I for advertisement optimization.

In its simplest form, advertisement optimization is an A/B/C testing problem where each advertisement is treated as an arm and the revenue generated constitutes the reward. In our example there are four possible advertisements to present. Initially, there exists no revenue history and arms are chosen at random. Over time, decisions are made and rewards are generated. Table I shows the input data for our running example with five advertisement decisions and the corresponding rewards. If

[1]https://github.com/fmr-llc/mabwiser

we want to exploit the most revenue-generating advertisement 90% of the time and explore the other options 10% of the time, we can use the ϵ-Greedy algorithm. The problem can then be modeled using MABWiser as follows:

```
1  arms      = [1, 2, 3, 4]
2  decisions = data['Ad']
3  rewards   = data['Revenue']
4  lp = LearningPolicy.EpsilonGreedy(epsilon=0.1)
5  bandit = MAB(arms, lp)
6  bandit.fit(decisions, rewards)
7  bandit.predict()
```

Given a set of advertisements to show, i.e., the arms (Line 1), the model is trained on the historic decisions and rewards (Line 2-3) using the *fit* method (Line 6) based on the greedy learning policy (Lines 4-5) with 10% randomization. When a new decision is needed, the *predict* method (Line 7) returns the next advertisement to present as determined by the learning policy.

This example assumes the best advertisement for one visitor is the same as for any other visitor. More likely, however, different advertisements will appeal to different people. We can thus use what we know about similar visitors, i.e. their contexts, to generate more accurate predictions. Suppose we know the visitor's age, past click rate, and whether they are an existing subscriber. Based on this, we can create a neighborhood of contexts to target our policy to learn only from similar visitors. The neighborhood policy extends the running example with contextual information:

```
8  contexts = data[['Age', 'ClickRate', 'Subscriber']]
9  np = NeighborhoodPolicy.Radius(radius=1)
10 bandit = MAB(arms, lp, np)
11 bandit.fit(decisions, rewards, contexts)
12 new_context = [[27, 0.2, 1]]
13 bandit.predict(new_context)
```

The *fit* method now has access to the historical context information for training (Lines 8 & 11). The bandit model (Line 10) learns a greedy policy from decision-reward pairs of similar contexts only. The *predict* method now operates on a given context (Line 13). Consequently, the next best advertisement is personalized for the context at hand.

Once a new decision is made, its corresponding reward is observed. This information can be used to update the bandit model. Suppose the previous recommendation was to show advertisement 3 which lead to \$15 in revenue. We can then update our model as follows:

```
14 bandit.partial_fit([3], [15], new_context)
```

The *partial_fit* method (Line 14) takes the same arguments as the *fit* method (Lines 6, 11) and updates the expectations of each arm with the additional information in preparation for the next prediction.

III. Background

A. Problem Definition

Multi-armed bandit problems address online sequential discrete decision making. In a MAB problem, each arm has an unknown model of stochastic or deterministic outcomes. The agent must make a sequence of decisions in time $1, 2, \ldots, T$. At each time t the agent is given a set of K arms, and decides which arm a_t to choose. It employs a "learning policy" to balance the exploration-exploitation trade-off when making decisions. After choosing an arm, a reward is observed that is associated with the selected arm $r_t \sim R^{a_t}$ while rewards for other arms remain unknown.

In many problems, we can also observe side information about the state of the environment at time t. This side information is referred to as *context* and is defined as $X_t \in \mathbb{R}^d$. The arm that has the highest expected reward $\mathbb{E}[r|a]$ may be different given different contexts. The uncertainty of the expected reward may also differ which can impact decisions when the exploration strategy leverages that information. The performance of a MAB is evaluated based on the cumulative reward given by $r = \sum_{t=1}^{T} r_t$ and the total regret, defined as the difference between the observed the reward r and the reward that would have been observed had the optimal arm been chosen in each decision r^*. The total regret, or total opportunity loss, can thus be represented as: $L = \mathbb{E}[\sum r^* - r]$. Overall, the objective of both context-free and contextual MABs is to maximize the cumulative reward, and equivalently, minimize the total regret.

B. Context-Free Bandits

We present a number of common strategies to drive exploration in context-free MABs [6], [7]:

ϵ-Greedy: Exploit the arm with the largest expected reward most of the time $(P(max) = 1 - \epsilon)$, but with probability ϵ explore a random arm.

Softmax: Select each arm with a probability proportionate to its mean reward as defined by the softmax distribution:

$$P(a_t) = \frac{e^{\frac{\mu_a}{\tau}}}{\Sigma e^{\frac{\mu}{\tau}}} \tag{1}$$

Thompson Sampling (TS): Rewards are categorized as successes or failures. The counts of the successes and failures for each arm are used to generate a beta distribution which is then sampled to obtain a probability of success. The arm with the highest sampled probability is chosen.

Upper Confidence Bound (UCB1): For each arm the Chernoff-Hoeffding bound is calculated based on the number of decisions that have been made and the arm with the highest upper confidence bound is selected [8]. The bounds are calculated as:

$$UCB_a = \mu_a + \alpha \times \sqrt{\frac{2 \times log(N)}{n_a}} \tag{2}$$

There are many variations on these in the literature such as ϵ-First [9], UCB2 and UCB-Normal [8].

C. Contextual Bandits

Contextual algorithms fall under two broad categories: parametric and non-parametric bandits.

Parametric Bandits: Parametric models include regression-based techniques such as LinUCB [2], LinTS [10], and LogUCB [11]. For example, given a context vector x of dimension d, the popular LinUCB algorithm trains a ridge regression

TABLE II
AVAILABLE BANDIT ALGORITHMS IN MABWISER

Algorithm	Parameters	Parameter Description
Random	-	-
ϵ-Greedy	epsilon	probability of random arm
Softmax	tau	scaling factor for exploration
Thompson Sampling	binarizer	function to binarize rewards
UCB1	alpha	exploration multiplier
LinUCB	alpha	exploration multiplier
	lambda	ridge regression lambda
K-Means	n_clusters	number of clusters
Clustering	is_minibatch	flag to use mini batch kmeans
K-Nearest	k	number of neighbors
Neighbors	metric	distance metric
Radius	radius	maximum distance
Neighbors	metric	distance metric

for each arm a and then calculates an upper confidence bound using the formula:

$$p_a = x\beta_a + \alpha\sqrt{(X_a^T X_a + \lambda * I_d)^{-1}x} \quad (3)$$

where α is the exploration factor, λ is the ridge regression strength, and X_a is the context history for the given arm.

Non-parametric Bandits: Non-parametric algorithms leverage context similarity to filter the training data and then a context-free learning policy is fit to this subset.

- **K-Means Clustering:** Uses k-means clustering to divide the training data into k clusters. Each cluster fits the learning policy separately, and at the predict step, the closest cluster is used for the given context [12].
- **K-Nearest Neighbors:** Selects the k nearest neighbors of the given context based on a distance metric. The learning policy targets only those selected neighbors [13].
- **Radius Neighbors:** Selects neighbors within the specified distance for the given context based on a distance metric. The learning policy targets only those selected neighbors.

IV. AVAILABLE BANDITS IN MABWISER

In MABWISER, context-free and parametric contextual bandits are available through a *learning policy*. Non-parametric contextual bandits are available through a *neighborhood policy* used in combination with a learning policy. Every bandit employs a learning policy while the neighborhood policy is optional. This design allows combining parametric policies with non-parametric ones, leading to hybrid strategies that are not yet well-studied in the research literature.

Table II presents the list of currently available bandits together with their parameters. We also include a random learning policy that selects an arm uniformly at random at each turn to serve as a baseline in simulations.

V. RELATED WORK

Despite the wide range of problems that can be modeled as CMABs, libraries and tools for this family of algorithms are limited in the Python stack as summarized in Table III.

For context-free algorithms, there are many options available for website A/B testing, including *Google Analytics* [14] and web-framework specific libraries. Facebook [15] and Yelp [16] both offer bandit-based tools for experiment optimization and hyper-parameter tuning of machine learning models. There are also more generalized Python libraries such as *SMPyBandits* [17] which, again, includes context-free algorithms only.

For CMAB algorithms, a few options are available:

- *Vowpal Wabbit* from Microsoft Research [18]. This is a C++ library with a substantial offering of online and offline machine learning algorithms including CMABs.
- *StreamingBandit* from Nth Iteration Labs at Tilburg University [19]. This is a Tornado webserver with a Python RESTful API for using CMABs in live applications.
- *Contextual* also from Nth Iteration Labs [20]. This R package is closest to ours with a more comprehensive set of learning policies. It provides context-free and contextual algorithms and supports online and offline experiments.
- *Deep Bayesian Bandits* from Google Brain [21]. This is a Python library that uses Tensorflow with a focus on applying Bayesian deep learning to CMAB problems.

Among these contextual bandit options, StreamingBandit and Deep Bayesian Bandits are available in Python. However, StreamingBandit only supports online learning in live field experiments whereas simulations are limited to offline and it requires a Tornado webserver, and Deep Bayesian Bandits is focused on deep learning research.

MABWISER fills the gap of a pure Python library with no external dependencies to provide access to fundamental context-free and contextual algorithms in both batch and online settings. In addition, we provide neighborhood-based non-parametric CMAB algorithms which are not available in any of the existing libraries.

VI. SYSTEM DESIGN

The architecture of the MABWISER library is presented in Figure 1. At a high-level, the design can be decomposed into three layers with increasing levels of complexity: the public interface layer (A), the integration layer (B), and the implementation layer (C).

A. Public Interface Layer

Users of the library interact with the MAB class shown in part A of Figure 1 to access the publicly available methods for training and testing. The ideas of training, testing, and online learning are embodied in the *fit*, *predict*, and *partial_fit* methods respectively. Data scientists and machine learning engineers who are familiar with the Python technology stack may have recognized the similarity between our library and the scikit-learn interface in our running example [5]. This allows bandits to behave similar to traditional machine learning models from the family of scikit-learn algorithms, as depicted on Lines 6-7 and Lines 11-13 in our running example, to facilitate ease of use and adoption.

A bandit model can be constructed using i) a set of arms, ii) a learning policy, and iii) an optional neighborhood policy,

TABLE III
COMPARISON OF EXISTING MULTI-ARMED BANDIT LIBRARIES

Library	Python	Non-Python Dependencies	Context-Free Bandits	Contextual Bandits	Custom Bandit Support	Simulation Utility
Google Analytics	+	+	+	-	-	-
Facebook Ax	+	+	+	-	+	+
Yelp MOE	+	+	+	-	-	-
SMPyBandits	+	-	+	-	-	+
Microsoft Vowpal Wabbit	-	+	+	+	-	+
StreamingBandit	+	+	+	+	+	+
Contextual	-	+	+	+	+	+
Google Deep Bayesian Bandits	+	-	-	+	+	+
MABWiser	+	-	+	+	+	+

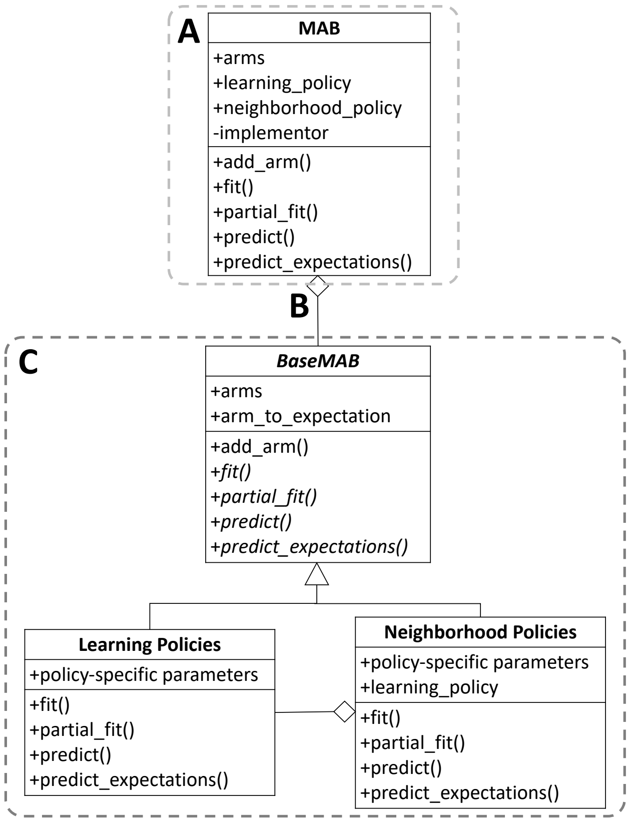

Fig. 1. The high-level architecture of the MABWISER library. A: Public Interface Layer. B: Integration Layer. C: Implementation Layer.

as depicted on Lines 5 and 10 in our earlier example. The learning policy and the neighborhood policy are named tuple objects. These immutable tuples serve as data containers for algorithm-specific parameters and help the user with drop-down selection, auto-completion and error highlighting when working within an integrated development environment.

We also provide a *predict_expectations* method which returns the expectation of each arm used in making predictions. This is similar to the predict probability method commonly found in machine learning classification models. The value of what we are calling the expectation, not to be confused with expected value, will differ depending on the learning policy. For a simple context-free ϵ-Greedy bandit the expectation is the mean reward of each arm, whereas for LinUCB it is the upper confidence bound of the expected reward of each arm for the given context, and for Thompson Sampling it is the sampled probability of success. Contextual bandits support batch prediction using multiple contexts as input while context-free bandits yield a single prediction at a time. As time proceeds, if new options become available the *add_arm* method enables introducing them to the model dynamically.

B. The Integration Layer

The MAB class given in Figure 1 also serves as the integration layer between the public interface in part A and the underlying implementation, the BaseMAB class, in part C. Specifically, the library uses the *bridge design pattern* [22] to decouple the interface from the details of algorithm implementation. The MAB class contains a pointer, or *implementor*, to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation based on the user-provided learning and neighborhood policies. The user interacts with the interface, and in turn, the interface delegates requests of training and testing to the implementor. Consequently, the interface object is the only "handle" known to the user while the implementation object is safely encapsulated. This not only simplifies the usage from a modeler perspective but also ensures that our library can continue to evolve in the future without breaking public use cases.

The integration layer serves two other purposes. For ease of use, it accepts multiple input types for decision, reward, and context data including lists, arrays, data frames, and pandas series. For efficiency, it examines the memory layout of these inputs and converts them to C-ordered row major numpy arrays if needed. If these low-level conversions were not handled properly, a seemingly naive operation like neighborhood calculation could easily become a significant runtime bottleneck.

C. The Implementation Layer

The BaseMAB class is the backbone of the implementation for every policy. It is a base class that handles boilerplate functionality. Each policy inherits from the class and specializes its training and testing components based on its learning algorithm. Practitioners with a general background in bandit algorithms can introduce new and custom policies by following the signature of the *fit* and *predict* methods which include type annotations and usage hints.

The implementation layer follows the principles of object-oriented design. For example, regression-based CMABs frequently calculate a standard regression and then perform exploration in the predict step. In our implementation of the LinUCB policy, there is a ridge regression with a configurable regularization strength following the steps recommended in [2] to initialize A with $\lambda \times I_d$ and β with zeroes and then update their values as data is fitted. This initialization allows arms with no historic data to still have coefficients to make predictions. The LinUCB policy inherits from the regression and modifies the prediction to calculate upper confidence bounds. Other parametric bandits such as LinTS and LogUCB follow the same pattern [10], [11]. Our design thus serves as a general blueprint for this greater class of algorithms.

VII. Parallelization

To accommodate large datasets and take advantage of modern multi-core architectures, the library provides built-in parallelization. During training, many of the operations performed for individual arms are independent and generally involve executing the same function on different subsets of the data filtered by arm. For context-free bandits, the fit step is parallelized across the arms for each of the learning policies. For contextual bandits, obtaining batch predictions for multiple contexts is an embarrassingly parallel problem. The set of contexts is distributed to multiple jobs that concurrently execute the prediction function on separate processors (cores). The parallelization uses the Python *joblib* package without requiring any external dependency.

The implementation details of the parallelization is handled in the BaseMAB class. The bandit policies focus on the unique operations: i) how to train a given arm, and ii) how to make a prediction given a context. Both operations are carried in parallel across arms and over sets of contexts.

Most of the learning policies include some form of randomization in their explore-exploit decision making process. As such, the reproducibility of results independent of how many cores are available or in which order tasks are allocated to different jobs becomes crucial. To guarantee repeatable results, special care is required to correctly seed each job. We follow a strategy suggested in [23]. First, a global random number generator is initiated with the user-provided or default seed. Then, in the master process the global random number generator yields additional seed values for each context vector. Each job can spawn a random number generator for each context if needed in their decision making while the overall process is guaranteed to be reproducible using the initial seed. This holds irrespective of the number of jobs or the order in which contexts are executed.

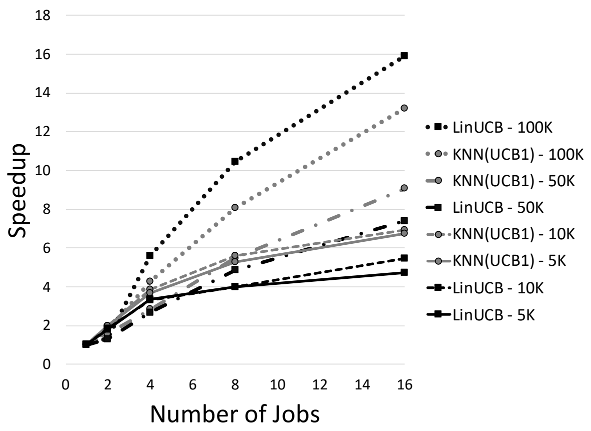

Fig. 2. Speedups achieved for the KNN(UCB1) and LinUCB contextual bandits as a function of number of jobs and the size of the dataset.

A. Performance Improvements

To quantify the benefits of our parallelization functionality, we ran a set of experiments to benchmark the speed-ups with one parametric policy, LinUCB, and one non-parametric policy, KNN(UCB1). We omitted context-free algorithms as they are sufficiently fast without parallelism.

We ran different sample sizes up to 200k with four context features and a 50-50 train-test split. We varied the number of jobs to be 1, 2, 4, 8 and 16. All experiments were run on a machine with 2 16-core Intel® Xeon® E5-2660 CPUs and 264 GB of RAM.

Figure 2 shows the results of these experiments. The speedups achieved by parallelism closely follows the increase in the dataset size. The increase in the number of jobs helps less if the dataset being processed is not large enough. For sufficiently large datasets, even super-linear speedups are possible thanks to the additional optimizations provided by numpy arrays. When combined with the simulator functionality described in the next section this makes performance comparison and parameter tuning possible on datasets with millions of observations using the above mentioned architecture.

VIII. Simulator & Hyper Parameter Tuning

Selecting the best bandit policy and identifying its optimal hyper-parameters for a given application requires running a large number of experiments on historical data sets. To facilitate this process, we provide a simulation utility to run concurrent experiments with multiple bandits. The simulator eliminates redundant code and tracks metrics about each bandit including confusion matrices of predicted versus historic decisions, and neighborhood statistics for neighborhood-based algorithms. The simulator automates the entire workflow from unscaled data to comprehensive evaluation metrics and performance plots. It supports online and offline simulations, chronological and random train-test splitting, and post-split normalization with a scaler provided by the user. These configurations are specified as simulation arguments:

```
bandits = [('1', MAB(arms, UCB1(1.0)),
           ('2', MAB(arms, LinUCB(1.0, 1.0))]
sim = Simulator(bandits, decisions, rewards,
18                  contexts, scaler=MinMaxScaler(),
19                  test_size=0.3, is_ordered=False,
20                  batch_size=0)
21 sim.run()
22 sim.plot(metric='avg')
```

In this scenario, we have two bandit policies of UCB1 and LinUCB to test on historical data (Lines 15-16). In addition to the algorithms available in the library, the Simulator natively supports custom bandit classes. The simulator receives the list of bandits, and the decision, reward and context data for training and testing (Lines 17-18). The user can set the fraction of data to include in the test set and the *is_ordered* flag controls whether to respect the in-place order or to split randomly in the train-test split (Line 19). The *batch_size* parameter triggers online learning when it is greater than zero (Line 20) to test and re-train at every batch within the test set.

Once the simulator is created, a simple *run* command starts the experiment (Line 21), throughout which metrics are collected and made accessible at the end. Once complete, a plot (Line 22) facilitates comparison of the bandits as well as each individual arm based on the specified level of optimism as a best-, average-, or worst-case scenario.

In addition to providing syntactic sugar, automation and metric tracking, the simulator makes performance improvements possible through lazy computation and shared data structures. For example, the distances between train and test data are the same across neighborhood bandits. These distances are calculated only once and shared thereafter.

An inherent challenge in the evaluation of any MAB algorithm with historical data is the rewards of the decisions that were not taken remain unknown. To accommodate this, for predictions that differ from the historic decisions, the simulator calculates best-, average-, worst-case reward estimates. In the simple case of a prediction matching the historic decision, the observed reward is used as-is. When a decision differs from the past, the maximum, mean and minimum reward for the predicted arm in the *training* data are used as proxies. For nearest neighbors-based policies, the neighborhood-specific training data is used for more accurate estimates. This evaluation component is configurable as an external function call to allow for application-specific evaluation.

IX. Conclusion

In this paper we introduced the design of MABWiser, an open-source library of multi-armed bandit algorithms for rapid prototyping and hyper-parameter tuning. The library addresses an immediate gap in the Python technology stack for the data science community. It provides both batch and online simulations for context-free, parametric and non-parametric contextual policies with built-in parallelization. As the library naturally supports custom extensions, other CMAB algorithms from the literature can be considered as potential future additions. We welcome the feedback of the community and hope our library can serve those working on bandit policies.

References

[1] J. Vermorel and M. Mohri, "Multi-armed bandit algorithms and empirical evaluation," in *Machine Learning: ECML 2005*, J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge, and L. Torgo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 437–448.

[2] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual-bandit approach to personalized news article recommendation," in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 661–670.

[3] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," in *Advances in neural information processing systems*, 2011, pp. 2249–2257.

[4] D. A. Berry, "Bayesian clinical trials," *Nature reviews Drug discovery*, vol. 5, no. 1, p. 27, 2006.

[5] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[6] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *CoRR*, vol. abs/1402.6028, 2014. [Online]. Available: http://arxiv.org/abs/1402.6028

[7] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen *et al.*, "A tutorial on thompson sampling," *Foundations and Trends® in Machine Learning*, vol. 11, no. 1, pp. 1–96, 2018.

[8] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[9] L. Tran-Thanh, A. Chapman, E. M. de Cote, A. Rogers, and N. R. Jennings, "Epsilon–first policies for budget–limited multi-armed bandits," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[10] S. Agrawal and N. Goyal, "Thompson sampling for contextual bandits with linear payoffs," in *International Conference on Machine Learning*, 2013, pp. 127–135.

[11] D. K. Mahajan, R. Rastogi, C. Tiwari, and A. Mitra, "Logucb: an explore-exploit algorithm for comments recommendation," in *Proceedings of the 21st ACM international conference on Information and knowledge management*. ACM, 2012, pp. 6–15.

[12] T. T. Nguyen and H. W. Lauw, "Dynamic clustering of contextual multi-armed bandits," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, ser. CIKM '14. New York, NY, USA: ACM, 2014, pp. 1959–1962. [Online]. Available: http://doi.acm.org/10.1145/2661829.2662063

[13] H. W. J. Reeve, J. Mellor, and G. Brown, "The k-nearest neighbour UCB algorithm for multi-armed bandits with covariates," *CoRR*, vol. abs/1803.00316, 2018. [Online]. Available: http://arxiv.org/abs/1803.00316

[14] Google, "Google analytics," 2019. [Online]. Available: https://marketingplatform.google.com/about/analytics/

[15] Facebook, Inc., "Ax: Adaptive experimentation platform," 2019. [Online]. Available: https://ax.dev

[16] Yelp, "Moe: Metric optimization engine," 2014. [Online]. Available: https://github.com/Yelp/MOE

[17] L. Besson, "Smpybandits: an open-source research framework for single and multi-players multi-arms bandits (mab) algorithms in python," 2018. [Online]. Available: https://GitHub.com/SMPyBandits/SMPyBandits/, documentation at https://SMPyBandits.GitHub.io

[18] J. Langford, L. Li, and A. Strehl, "Vowpal wabbit," 2007. [Online]. Available: https://github.com/VowpalWabbit/vowpal_wabbit/wiki

[19] J. Kruijswijk, R. van Emden, P. Parvinen, and M. Kaptein, "Streamingbandit; experimenting with bandit policies," *arXiv preprint arXiv:1602.06700*, 2016.

[20] R. van Emden and M. Kaptein, "contextual: Evaluating contextual multi-armed bandit problems in R," *CoRR*, vol. abs/1811.01926, 2018. [Online]. Available: http://arxiv.org/abs/1811.01926

[21] C. Riquelme, G. Tucker, and J. Snoek, "Deep bayesian bandits showdown," in *International Conference on Learning Representations*, 2018.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson, 2016.

[23] A. A. Ciré, S. Kadioglu, and M. Sellmann, "Parallel restarted search," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 2014, pp. 842–848. [Online]. Available: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8597