SMPyBandits, a Research Framework for Single and Multi-Players Multi-Arms Bandits Algorithms in Python

Lilian Besson*

February 28, 2018

Abstract

I present the open-source numerical environment *SMPyBandits*, written in Python and designed to be an easy to use framework for experimenting with single- and multi-player algorithms and in different variations of the multi-armed bandits problem.

Contents

* ✉: Lilian.Besson[AT]CentraleSupelec[.]fr, ORCID: 0000-0003-2767-2563
PhD Student at CentraleSupélec, campus of Rennes, SCEE team & Inria Lille Nord Europe, SequeL team. Thanks to Emilie Kaufmann and Christophe Moy for their review and support.

1 Summary

This article presents my numerical environment *SMPyBandits*, written in Python (2 or 3) [Fou17], for numerical simulations on *single*-player and *multi*-players Multi-Armed Bandits (MAB) algorithms [BCB12].

SMPyBandits is the most complete open-source implementation of state-of-the-art algorithms tackling various kinds of sequential learning problems referred to as Multi-Armed Bandits. It aims at being extensive, simple to use and maintain, with a clean and perfectly documented codebase. But most of all it allows fast prototyping of simulations and experiments, with an easy configuration system and command-line options to customize experiments while starting them (see below for an example).

SMPyBandits does not aim at being blazing fast or perfectly memory efficient, and comes with a pure Python implementation with no dependency except standard open-source Python packages. Even if some critical parts are also available as a `C` Python extension, and even by using Numba [I+17] whenever it is possible, if simulation speed really matters, one should rather refer to less exhaustive but faster implementations, like for example [Lat16] in `C++` or [Raj17] in Julia.

1.1 Presentation

1.1.1 Single-Player MAB

Multi-Armed Bandit (MAB) problems are well-studied sequential decision making problems in which an agent repeatedly chooses an action (the "arm" of a one-armed bandit) in order to maximize some total reward [Rob52, LaiRobbins85]. Initial motivation for their study came from the modeling of clinical trials, as early as 1933 with the seminal work of Thompson [Tho33]. In this example, arms correspond to different treatments with unknown, random effect. Since then, MAB models have been proved useful for many more applications, that range from cognitive radio [JEMP09] to online content optimization (news article recommendation [LCLS10], online advertising [CL11] or A/B Testing [**(author?)** [KCG14];Jamieson17]), or portfolio optimization [SLM12].

This Python package is the most complete open-source implementation of single-player (classical) bandit algorithms (over 65!). We use a well-designed hierarchical structure and class inheritance scheme to minimize redundancy in the codebase, and for instance the code specific to the UCB algorithm [LR85, ACBF02] is as short as this (and fully documented), by inheriting from the `IndexPolicy` class:

```python
from numpy import sqrt, log
from .IndexPolicy import IndexPolicy

class UCB(IndexPolicy):
  """ The UCB policy for bounded bandits.
  Reference: [Lai & Robbins, 1985]. """

  def computeIndex(self, arm):
    r""" Compute the current index, at time t and
    after :math:`N_k(t)` pulls of arm k:

    .. math::

        I_k(t) = \frac{X_k(t)}{N_k(t)}
        + \sqrt{\frac{2 \log(t)}{N_k(t)}}.
    """
    if self.pulls[arm] < 1:  # forced exploration
      return float('+inf')   # in the first steps
    else:                    # or compute UCB index
      estimated_mean = (self.rewards[arm] / self.pulls[arm])
      exploration_bias = sqrt((2 * log(self.t)) / self.pulls[arm])
      return estimated_mean + exploration_bias
```

1.1.2 Multi-Players MAB

For Cognitive Radio applications, a well-studied extension is to consider $M \geq 2$ players, interacting on the *same* K arms. Whenever two or more players select the same arm at the same time, they all suffer from a collision. Different collision models has been proposed, and the simplest one consist in giving a 0 reward to each colliding players. Without any centralized supervision or coordination between players, they must learn to access the M best resources (*i.e.*, arms with highest means) without collisions.

This package implements all the collision models found in the literature, as well as all the algorithms from the last 10 years or so (including `rhoRand` from 2009, `MEGA` from 2015, `MusicalChair` from 2016, and our state-of-the-art algorithms `RandTopM` and `MCTopM`) from [BK18a].

1.2 Purpose

The main goal of this package is to implement with the same API most of the existing single- and multi-player multi-armed bandit algorithms. Each algorithm comes with a clean documentation page, containing a reference to the research article(s) that introduced it, and with remarks on its numerical efficiency.

It is neither the first nor the only open-source implementation of multi-armed bandits algorithms, although one can notice the absence of any well-maintained reference implementation. I built *SMPyBandits* from a framework called *pymaBandits* [CGK12], which implemented a few algorithms and three kinds of arms, in both Python and MATLAB. The goal was twofolds, first to implement as many algorithms as possible to have a complete implementation of the current state of research in MAB, and second to implement multi-players simulations with different models.

Since November 2016, I follow actively the latest publications related to Multi-Armed Bandits (MAB) research, and usually I implement quickly any new algorithms. For instance, Exp3++, CORRAL and SparseUCB were each introduced by articles (for Exp3++, for CORRAL, for SparseUCB) presented at COLT in July 2017, LearnExp comes from a NIPS 2017 paper, and kl-UCB++ from an ALT 2017 paper.

1.3 Features

With this numerical framework, simulations can run on a single CPU or a multi-core machine using joblib [Var17], and summary plots are automatically saved as high-quality PNG, PDF and EPS (ready for being used in research article), using matplotlib [Hun07] and seaborn [W+17]. Making new simulations is very easy, one only needs to write a configuration script and no knowledge of the internal code architecture.

1.3.1 Examples of configuration for some simulations

A small script `configuration.py` is used to import the arm classes, the policy classes and define the problems and the experiments. For instance, we can compare the standard anytime `klUCB` algorithm against the non-anytime variant `klUCBPlusPlus` algorithm, as well as `UCB` (with $\alpha = 1$) and `Thompson` (with Beta posterior). See below in Figure 1 for the result showing the average regret for these 4 algorithms.

```
from Arms import *;  from Policies import *

configuration = {
```

```
  "horizon": 10000,      # Finite horizon of the simulation
  "repetitions": 1000,   # Number of repetitions
  "n_jobs": -1,          # Max number of cores for parallelization
  # Environment configuration, you can set up more than one.
  "environment": [ {
      "arm_type": Bernoulli,
      "params": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
  }],
  # Policies that should be simulated, and their parameters.
  "policies": [
      {"archtype": klUCB, "params": {} },
      {"archtype": klUCBPlusPlus,
       "params": { "horizon": 10000 } },
      {"archtype": UCBalpha,
       "params": { "alpha": 1 } },
      {"archtype": Thompson, "params": {} },
  ]}
```

For a second example, this snippet is a minimal example[1] of configuration for multiplayer simulations, comparing different multi-player algorithms used with the `klUCB` index policy. See below in Figure 2 for an illustration.

```
from Arms import *;  from Policies import *
from PoliciesMultiPlayers import *
nbPlayers = 3

configuration = {
  "horizon": 10000,     # Finite horizon of the simulation
  "repetitions": 100,   # Number of repetitions
  "n_jobs": -1,         # Max number of cores for parallelization
  # Environment configuration, you can set up more than one.
  "environment": [ {
      "arm_type": Bernoulli,
      "params": [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
  } ],
  # Policies that should be simulated, and their parameters.
  "successive_players": [
      CentralizedMultiplePlay(nbPlayers, nbArms, klUCB).children,
      RandTopM(nbPlayers, nbArms, klUCB).children,
      MCTopM(nbPlayers, nbArms, klUCB).children,
      Selfish(nbPlayers, nbArms, klUCB).children,
```

[1] See the file `configuration_multiplayers.py` in the code for more details.

```
        rhoRand(nbPlayers, nbArms, klUCB).children,
    ] }
```

1.3.2 Documentation

A complete sphinx [B+18] documentation for each algorithms and every piece of code, included the constants in the different configuration files, is available here: http://banditslilian.gforge.inria.fr.

1.3.3 Other noticeable features

1.3.3.1 Single-player Policies

- More than 65 algorithms, including all known variants of the `UCB`, kl-UCB, `MOSS` and Thompson Sampling algorithms, as well as other less known algorithms (`OCUCB`, `BESA`, `OSSB` etc).
- Implementation of very recent Multi-Armed Bandits algorithms, e.g., `kl-UCB++`, `UCB-dagger`, or `MOSS-anytime` (from this COLT 2016 article).
- Experimental policies: `BlackBoxOpt` or `UnsupervisedLearning` (using Gaussian processes to learn the arms distributions).

1.3.3.2 Arms and problems

- The framework mainly targets stochastic bandits, with arms following `Bernoulli`, bounded (truncated) or unbounded `Gaussian`, `Exponential`, `Gamma` or `Poisson` distributions.
- The default configuration is to use a fixed problem for N repetitions (e.g. 1000 repetitions, use `MAB.MAB`), but there is also a perfect support for "Bayesian" problems where the mean vector $\mu_1, \ldots, \mu_K$ change *at every repetition* (see `MAB.DynamicMAB`).
- There is also a good support for Markovian problems, see `MAB.MarkovianMAB`, even though I preferred to not implement policies specifically designed for Markovian problems.

1.4 Other remarks

- The framework is implemented in an imperative and object oriented style. Algorithm and arms are represented as classes, and the API of the `Arms`, `Policy` and `MultiPlayersPolicy` classes is clearly documented.

- The code is clean, and a special care is given to keep it compatible for both Python 2 and Python 3.
- The joblib library [Var17] is used for the `Evaluator` classes, so the simulations are easily ran in parallel on multi-core machines and servers.[2]

1.4.1 How to run the experiments ?

For example, this short bash snippet[3] shows how to clone the code, install the requirements for Python 3 (in a virtualenv [BPP16]), and starts some simulation for $N = 1000$ repetitions of the default non-Bayesian Bernoulli-distributed problem, for $K = 9$ arms, an horizon of $T = 10000$ and on 4 CPUs.[4] Using environment variables (`N=1000`) when launching the simulation is not required but it is convenient.

```
# 1. get the code in /tmp/, or wherever you want
cd /tmp/
git clone https://GitHub.com/SMPyBandits/SMPyBandits.git
cd SMPyBandits.git
# 2. just be sure you have the latest virtualenv from Python 3
sudo pip3 install --upgrade virtualenv
# 3. create and active the virtualenv
virtualenv3 venv || virtualenv venv
. venv/bin/activate
# 4. install the requirements in the virtualenv
pip3 install -r requirements.txt
# 5. run a single-player simulation!
N=1000 T=10000 K=9 N_JOBS=4 make single
# 6. run a multi-player simulation for 3 players!
N=1000 T=10000 M=3 K=9 N_JOBS=4 make moremulti
```

1.4.2 Examples of illustrations

The two simulations above produce these plots showing the average cumulated regret[5] for each algorithm, which is the reference measure of efficiency for algorithms in the multi-armed bandits framework.

---

[2] Note that *SMPyBandits* does no need a GPU and is not optimized to run on a cluster. In particular, it does not take advantage of popular libraries like numexpr, theano or tensorflow.

[3] See this page of the documentation for more details.

[4] It takes about 20 to 40 minutes for each simulation, on a standard 4-cores 64 bits GNU/Linux laptop.

[5] The regret is the difference between the cumulated rewards of the best fixed-armed strategy (which is the oracle strategy for stationary bandits) and the cumulated rewards of the considered algorithms.

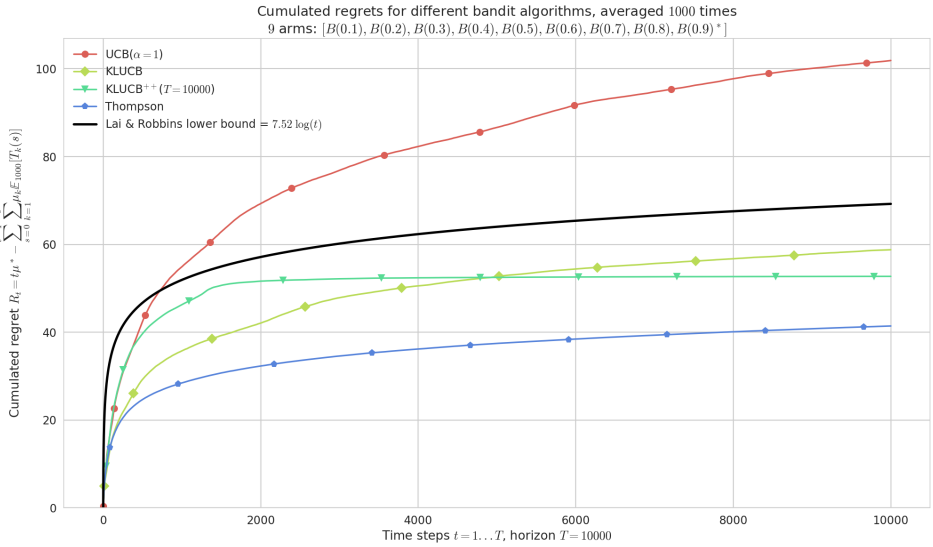

Figure 1: Single-player simulation showing the regret of 4 algorithms, and the asymptotic lower-bound from [LR85]. They all perform very well, and at finite time they are empirically *below* the asymptotic lower-bound. Each algorithm is known to be order-optimal (*i.e.*, its regret is proved to match the lower-bound up-to a constant), and each but UCB is known to be optimal (*i.e.* with the constant matching the lower-bound).

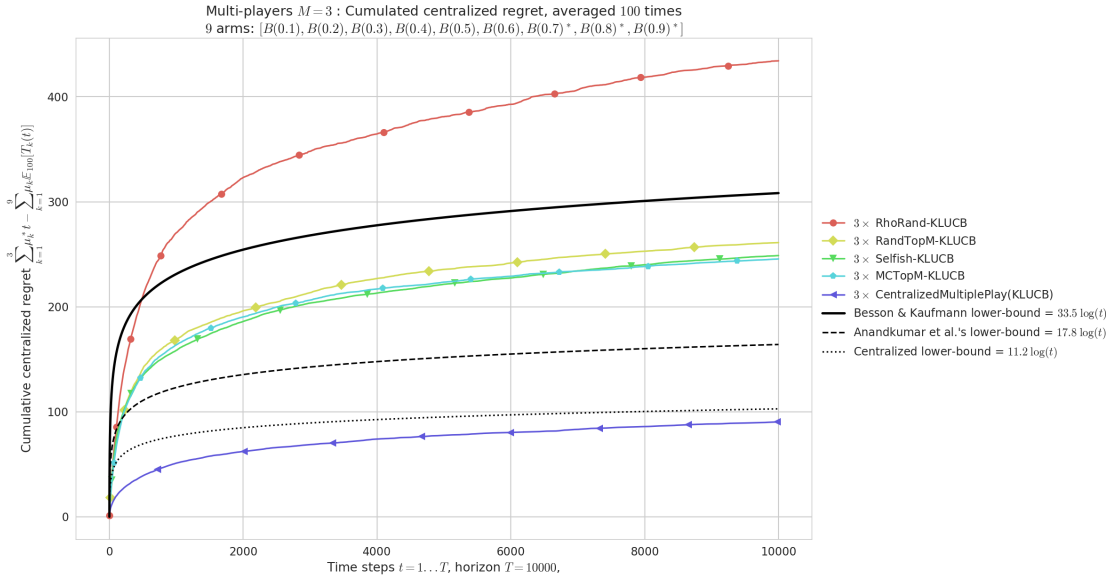

Figure 2: Multi-player simulation showing the regret of 6 algorithms, and the asymptotic lower-bound from [BK18a]. The best algorithm is the centralized version, but for decentralized algorithms, our proposals outperform the previous state-of-the-art `rhoRand` policy.

1.5 Research using *SMPyBandits*

SMPyBandits was used for the following research articles since 2017:[6]

- For this first article, [BBM+17], *SMPyBandits* was not used to generate the main figures, but to explore on a smaller scale many other approaches (using `EvaluatorSparseMultiPlayers`).
- For [BK18a], we used *SMPyBandits* for all the simulations for multi-player bandit algorithms.[7] We designed the two `RandTopM` and `MCTopM` algorithms and proved than they enjoy logarithmic regret in the usual setting, and outperform significantly the previous state-of-the-art solutions (*i.e.*, `rhoRand`, `MEGA` and `MusicalChair`).
- In [BKM18], we used *SMPyBandits* to illustrate and compare different aggregation algorithms.[8] We designed a variant of the Exp3 algorithm for online aggregation of experts [BCB12], called `Aggregator`. Aggregating experts is a well-studied idea in

[6] I (Lilian Besson) have started my PhD in October 2016, and this is a part of my **on going** research since December 2016. I launched the documentation on March 2017, I wrote my first research articles using this framework in 2017 and I was finally able to open-source my project in February 2018.

[7] More details and illustrations are given on the documentation page, `MultiPlayers`.

[8] More details and illustrations are given on the documentation page, `Aggregation`.

sequential learning and in machine learning in general. We showed that it can be used in practice to select on the run the best bandit algorithm for a certain problem from a fixed pool of experts. This idea and algorithm can have interesting impact for Opportunistic Spectrum Access applications [JEMP09] that use multi-armed bandits algorithms for sequential learning and network efficiency optimization.

- In [BK18b], we used *SMPyBandits* to illustrate and compare different "doubling trick" schemes.[9] In sequential learning, an algorithm is *anytime* if it does not need to know the horizon T of the experiments. A well-known trick for transforming any non-anytime algorithm to an anytime variant is the "Doubling Trick": start with an horizon $T_0 \in \mathbb{N}$, and when $t > T_i$, use $T_{i+1} = 2T_i$. We studied two generic sequences of growing horizons (geometric and exponential), and we proved two theorems that generalized previous results. A geometric sequence suffices to minimax regret bounds (in $R_T = \mathcal{O}(\sqrt{(T)})$), with a constant multiplicative loss $\ell \leq 4$, but cannot be used to conserve a logarithmic regret bound (in $R_T = \mathcal{O}(\log(T))$). And an exponential sequence can be used to conserve logarithmic bounds, with a constant multiplicative loss also $\ell \leq 4$ in the usual setting. It is still an open question to know if a well-tuned exponential sequence can conserve minimax bounds or weak minimax bounds (in $R_T = \mathcal{O}(\sqrt{T \log(T)})$).

1.6 Dependencies

The framework is written in Python [Fou17], using matplotlib [Hun07] for 2D plotting, numpy [vdWCV11] for data storing, random number generations and and operations on arrays, scipy [JOP+] for statistical and special functions, and seaborn [W+17] for pretty plotting and colorblind-aware colormaps. Optional dependencies include joblib [Var17] for parallel simulations, numba [I+17] for automatic speed-up on some small functions, as well as sphinx [B+18] for generating the documentations. I also acknowledge the use of virtualenv [BPP16] for launching simulations in isolated environments, and jupyter [K+16] used with ipython [PG07] to experiment with the code.

References

[ACBF02] Peter Auer, Nicolò Cesa-Bianchi, and Peter Fischer. Finite-time Analysis of the Multi-armed Bandit Problem. *Machine Learning*, 47(2):235–256, 2002.

[B+18] Georg Brandl et al. Sphinx: Python documentation generator. Online at: `www.sphinx-doc.org`, Februry 2018.

[9] More details and illustrations are given on the documentation page, `DoublingTrick`.

[BBM+17] Rémi Bonnefoi, Lilian Besson, Christophe Moy, Émilie Kaufmann, and Jacques Palicot. Multi-Armed Bandit Learning in IoT Networks: Learning helps even in non-stationary settings. In *12th EAI Conference on Cognitive Radio Oriented Wireless Network and Communication*, CROWNCOM Proceedings, 2017.

[BCB12] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret Analysis of Stochastic and Non-Stochastic Multi-Armed Bandit Problems. *Foundations and Trends® in Machine Learning*, 5(1), 2012.

[BK18a] Lilian Besson and Emilie Kaufmann. Multi-Player Bandits Models Revisited. In *Algorithmic Learning Theory*, Lanzarote, Spain, April 2018.

[BK18b] Lilian Besson and Emilie Kaufmann. What Doubling Trick Can and Can't Do for Multi-Armed Bandits. working paper or preprint, February 2018.

[BKM18] Lilian Besson, Emilie Kaufmann, and Christophe Moy. Aggregation of Multi-Armed Bandits Learning Algorithms for Opportunistic Spectrum Access. In *IEEE WCNC - IEEE Wireless Communications and Networking Conference*, Barcelona, Spain, April 2018.

[BPP16] Ian Bicking, The Open Planning Project, and PyPA. Virtualenv: a tool to create isolated Python environments. Online at: `virtualenv.pypa.io/en/stable`, November 2016.

[CGK12] Olivier Cappé, Aurélien Garivier, and Emilie Kaufmann. pymaBandits. Online at: `mloss.org/software/view/415`, 2012.

[CL11] Olivier Chapelle and Lihong Li. An Empirical Evaluation of Thompson Sampling. In *Advances in Neural Information Processing Systems*, pages 2249–2257. Curran Associates, Inc., 2011.

[Fou17] Python Software Foundation. Python language reference, version 3.6. Online at: `www.python.org`, October 2017.

[Hun07] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[I+17] Anaconda Inc. et al. Numba, NumPy aware dynamic Python compiler using LLVM. Online at: `numba.pydata.org`, 2017.

[JEMP09] Wassim Jouini, Daniel Ernst, Christophe Moy, and Jacques Palicot. Multi-Armed Bandit Based Policies for Cognitive Radio's Decision Making Issues. In *International Conference Signals, Circuits and Systems*. IEEE, 2009.

[JOP+] Eric Jones, Travis E. Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. Online at: `www.scipy.org`, 2001–.

[K+16] Thomas Kluyver et al. Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87–90. IOS Press, 2016.

[KCG14] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the Complexity of A/B Testing. In *Conference on Learning Theory*, pages 461–481. PMLR, 2014.

[Lat16] Tor Lattimore. Library for Multi-Armed Bandit Algorithms. Online at: `github.com/tor/libbandit`, 2016.

[LCLS10] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *International Conference on World Wide Web*, pages 661–670. ACM, 2010.

[LR85] T. L. Lai and Herbert Robbins. Asymptotically Efficient Adaptive Allocation Rules. *Advances in Applied Mathematics*, 6(1):4–22, 1985.

[PG07] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[Raj17] Vishnu Raj. A Julia Package for providing Multi Armed Bandit Experiments. Online at: `github.com/v-i-s-h/MAB.jl`, 2017.

[Rob52] Herbert Robbins. Some Aspects of the Sequential Design of Experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[SLM12] Amir Sani, Alessandro Lazaric, and Rémi Munos. Risk-Aversion In Multi-Armed Bandits. In *Advances in Neural Information Processing Systems*, pages 3275–3283, 2012.

[Tho33] William R. Thompson. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25, 1933.

[Var17] Gaël Varoquaux. Joblib: running Python functions as pipeline jobs. Online at: `pythonhosted.org/joblib`, March 2017.

[vdWCV11] Stéfan van der Walt, Chris S. Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, March 2011.

[W+17] Michael Waskom et al. Seaborn: statistical data visualization. Online at: `seaborn.pydata.org`, September 2017.