Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. doi: 10.18637/jss.v000.i00

contextual: Simulating Contextual Multi-Armed Bandit Problems in R

Robin van Emden
JADS

Maurits Kaptein
Tilburg University

Abstract

Over the past decade, contextual bandit algorithms have been gaining in popularity due to their effectiveness and flexibility in the evaluation of sequential decision problems - from online advertising and recommender systems to clinical trial design and personalized medicine. At the same time, there are as of yet surprisingly few options that enable researchers and practitioners to simulate and compare the wealth of new and existing bandit algorithms in a standardized way. To help close this gap between analytical research and practical evaluation the current paper introduces the object-oriented R package **contextual**: a user-friendly and, through its object-oriented structure, easily extensible framework that facilitates parallelized comparison of contextual and non-contextual Bandit policies through both simulation and offline analysis.

Keywords: contextual multi-armed bandits, simulation, sequential experimentation, R.

1. Introduction

There are many real-world situations in which we have to decide between a set of options and have to learn the best course of action by choosing one way or the other sequentially, learning one step at a time. In such situations, the basic premise is the same for each and every renewed decision: do you stick to what you already know and receive an expected result ("exploit") or choose an option you do not know all that much about and potentially learn something new ("explore")? As we all encounter such dilemma's on a daily basis (Wilson, Geana, White, Ludvig, and Cohen 2014), it is easy to come up with examples - for instance:

- When going out to dinner, do you explore new restaurants, or do you exploit familiar ones?
- As an online marketer, do you try a new ad, or keep the current one?
- As a doctor, do you treat your patients with tried and tested medication, or do you prescribe a new and promising experimental treatment?

Or maybe you received some gambling money, and now have to decide which out of k slot machines or "one-armed bandits"[1] to play[2]. Each "bandit" is similar but differs subtly in its average payout over time. So you need to come up with a strategy or "policy" for which bandit's arm to play. One option would be to try every arm once and then go with the arm that offered you the biggest winnings. However, those winnings might have been just a lucky fluke. On the other hand, every time you explore another arm, you may potentially lose out compared to the arm that had been doing well up till then—a classic balancing act known as the "explore-exploit dilemma". To get a better grip on this set of decision problems, and to learn if and when specific strategies might be more successful than others, such explore/exploit dilemmas have been studied extensively since the 1930s[3] under the umbrella of the "Multi-Armed Bandit" (MAB) problem (Auer, Cesa-Bianchi, and Fischer 2002; Lai and Robbins 1985; Bubeck, Cesa-Bianchi *et al.* 2012). todo: auer and lai elsewhere

A recent MAB generalization known as the *contextual* Multi-Armed Bandit (cMAB) builds on the previous by adding one crucial element: contextual information (Langford and Zhang 2008). These contextual multi-armed bandits are known by many different names in about as many different fields of research (Tewari and Murphy 2017)—for example as "bandit problems with side observations" (Wang, Kulkarni, and Poor 2005), "bandit problems with side information" (Lu, Pál, and Pál 2010), "associative reinforcement learning" (Kaelbling, Littman, and Moore 1996), "reinforcement learning with immediate reward" (Abe, Biermann, and Long 2003), "associative bandit problems" (Strehl, Mesterharm, Littman, and Hirsh 2006), or "bandit problems with covariates" (Sarkar 1991). However, the term "contextual Multi-Armed Bandit," as conceived by Langford and Zhang (2008), is the most generally used—so that is the term we will use in the current paper.

Still, however named, all cMAB policies differentiate themselves, by definition, from their MAB cousins in that they are able to make use of features that reflect the current state of the world—features that can then be mapped onto available arms or actions[4]. This access to side information makes cMAB algorithms yet more relevant to many real-life decision problems than their MAB progenitors (Langford and Zhang 2008). To follow up on our previous examples: do you show a particular add to returning customers, to new ones, or both? Do you prescribe a different treatment to male patients, female patients, or both? In the real world, it appears no choice exists without some contextual information to be mined or mapped. So it may be no surprise that there is interest in the application of cMAB algorithms in many different areas: from recommendation engines (Lai and Robbins 1985) to advertising (Tang, Rosales, Singh, and Agarwal 2013) and (personalized) medicine Tewari and Murphy (2017)—inspiring a multitude of new, often analytically derived bandit algorithms or policies, each with their strengths and weaknesses.

Regrettably, though cMAB algorithms have gained both academic and commercial acclaim, comparisons on simulated, and, importantly, real-life, large-scale offline “partial label” data sets have relatively lagged behind. To this end, the current paper introduces the **contextual** R package. **contextual** aims to facilitate the simulation, offline comparison, and evaluation of (Contextual) Multi-Armed bandit

[1]The term “one-armed bandit” actually refers to a type of slot machine found in casino’s, where the “arm” is a lever that the gambler pulls on feeding it a quarter.

[2]Or one big slot machine or "bandit" with k arms, which is how this problem would be formalized in the Multi-Armed Bandit literature.

[3]As Dr. Peter Whittle famously stated "[the problem] was formulated during the [second world] war, and efforts to solve it so sapped the energies and minds of Allied analysts that the suggestion was made that the problem be dropped over Germany, as the ultimate instrument of intellectual sabotage." (Whittle 1979)

[4]That is, before making a choice, the learner receives information on the state of the world or "context" in the form of a d-dimensional feature vector. After making a choice the learner is then able to combine this contextual information with the reward received to make a more informed decision in the next round.

policies. There do exist a few other frameworks that enable the analysis and comparison of either on- or offline datasets in some capacity, such as Microsoft's Vowpal Wabbit (Langford, Li, and Strehl 2007), the online evaluation platform StreamingBandit (Kaptein and Kruijswijk 2016), and the MAB focussed python packages Striatum (201 2018) and SMPyBandits (Besson 2018). But, as of yet, no extensible and widely applicable R (R Core Team) package that can analyze and compare, respectively, K-armed, Continuum and Contextual Multi-Armed Bandit Algorithms on both simulated and offline data.

In section 2, this paper continues with a more formal definition of MAB and CMAB problems and relate it to our implementation. In section 3, we give an overview of **contextual**'s object-oriented structure In section 4, we list the policies that are available by default, and simulate two MAB policies and a cMAB policy. In section 5, we demonstrate how easy it is to extend and customize **contextual** policies and bandits. In section 6, we replicate two papers, thereby demonstrating how to test policies on offline data sets. Finally, in section 7, we will go over some of the additional features in the package and conclude with some comments on the current state of the package and possible enhancements.

2. From formalization to implementation

In the current section, we first present a more formal definition of the contextual Multi-Armed Bandit problem. We then show how this formalization can be translated to a clear and concise class structure.

2.1. Formalization

On further formalization of the contextual Bandit problem, a (k-armed) **bandit** B can be defined as a set of k distributions $B = \{D_1, \dots, D_k\}$, where each distribution is associated with the I.I.D. rewards generated by one of the $k \in \mathbb{N}^+$ arms. We now define an algorithm or **policy** π, that seeks to maximize its total **reward** (that is, to maximize its cumulative reward $\sum_{t=1}^{T} r_t$ or minimize its cumulative regret—see equations 1, 2 and 3). This **policy** observes information on the current state of the world represented as a d-dimensional contextual feature vector $x_t = (x_{1,t}, \dots, x_{d,t})$. Based on earlier payoffs, the **policy** then selects one of **bandit** B's arms by choosing an action $a_t \in \{1, \dots, k\}$, and receives reward $r_{a_t,t}$, the expectation of which depends both the context and the reward history of that particular arm. With this observation $(x_{t,a_t}, a_t, r_{t,a_t})$, the policy now updates its arm-selection strategy.

In practice, for scalability reasons, **policies** generally use a limited set of parameters θ_t. This set of paramaters summarizes all historical interactions $H_{t'} = (x_{t,a_t}, a_t, r_{t,a_t})$ over t= {1, . . . , t'}, ensuring that the dimensionality of $\theta_{t'} << H_{t'}$.

These steps are then repeated T times, where T is generally defined as a bandit's **horizon**.

Schematically, for each round t= {1, . . . , T}:

1) Policy π observes state of the world as contextual feature vector $x_t = (x_{1,t}, \dots, x_{d,t})$
2) Bandit B generates reward vector $r_t = (r_{t,1}, \dots, r_{t,k})$
3) Policy π selects one of bandit B's arms $a_t \in \{1, \dots, k\}$
4) Policy π gets reward r_{t,a_t} from bandit B and updates its arm-selection strategy with $(x_{t,a_t}, a_t, r_{t,a_t})$

The goal of the policy π is to optimize its *cumulative reward* over t= { 1, . . . , T }

$$Reward_T^{\pi} = \sum_{t=1}^{T} (r_{a_t^{\pi}, x_t}) \tag{1}$$

Though *cumulative reward* offers a first estimate of a policy's learning performance, as a performance measure, *cumulative regret*—defined as the sum of rewards that would have been received by choosing optimal actions a at every t subtracted by the sum of rewards awarded to the actually chosen actions a^{π} for every t over t= { 1, . . . , T }—offers several advantages.

Firstly, with cumulative regret, you can shift a bandit's rewards by some arbitrary constant, and still arrive at the same total cumulative regret over T.

Secondly, as *cumulative regret* grows only on selecting non-optimal arms, a good policy's cumulative regret ought to be growing less and less over T.

$$R_T^{\pi} = \max_{\mathrm{a}=1,\ldots,k} \sum_{t=1}^{T} (r_{\mathrm{a}, x_t}) - \sum_{t=1}^{T} (r_{a_t^{\pi}, x_t}) \tag{2}$$

See for example Figure **??** in section 4.3 for an illustrative example of how cumulative regret tends to work better as a performance measure than cumulative reward. Or, in practice, policies' *expected* cumulative regret:

$$\mathbb{E}\left[R_T^{\pi}\right] = \mathbb{E}\left[\max_{\mathrm{a}=1,\ldots,k} \sum_{t=1}^{T} (r_{\mathrm{a}, x_t}) - \sum_{t=1}^{T} (r_{a_t^{\pi}, x_t})\right] \tag{3}$$

As expectation $\mathbb{E}[\cdot]$ is generally taken with respect to random draws of both rewards assigned by a bandit and arms as selected by a policy.

2.2. Basic Implementation

We set out to develop an implementation that stays close to the previous formalization while offering maximum flexibility and extensibility. As a bonus, this kept the class structure of the package elegant and straightforward, with six classes forming the backbone of the package (see also Figure 2.2):

- `Bandit`: The R6 class `Bandit` is the parent class of all `Bandits` implemented in **contextual**. Classes that extend the abstract superclass `Bandit` are responsible for both the generation of `d` dimensional `context` vectors `X` and the `k` I.I.D. distributions each generating a `reward` for each of its `k` arms at each time step `t`. `Bandit` subclasses can (pre)generate these values synthetically, based on offline data, etc.
- `Policy`: The R6 class `Policy` is the parent class of all `Policy` implementations in **contextual**. Classes that extend this abstract `Policy` superclass are expected to take into account the current `d` dimensional `context`, together with a limited set of parameters denoted `theta` (summarizing all past `contexts`, `actions` and `rewards`), to choose one of a `Bandit`'s arms at each time step `t`. On choosing one of the k arms of the Bandit and receiving its corresponding `reward`, the `Policy` then uses the current `context`, `action` and `reward` to update its set of parameters `theta` (which summarize all historical interactions).

- `Agent`: The R6 class `Agent` is responsible for the state, flow of information between and the running of one `Bandit`/`Policy` pair. As such, multiple `Agents` can be run in parallel with each separate Agent keeping track of `t` and the parameters in `theta` for its assigned `Policy` and `Bandit` pair.
- `Simulator`: The R6 class `Simulator` is the entry point of any **contextual** simulation. It encapsulates one or more `Agents` (in parallel, by default), clones them if necessary, runs the `Agents`, and saves the log of all of the `Agents` interactions to a `History` object.
- `History`: The R6 class `History` keeps a log of all `Simulator` interactions in its internal `data.table`. It also provides basic data summaries, and can save and load simulation data.
- `Plot`: The R6 class `Plot` generates plots based on `History` data. It is usually actually invoked by calling the generic function `plot(h)`, where `h` is an `History` class instance.

From these building blocks, we are now able to put together a basic five line MAB simulation:

```
policy      <- EpsilonGreedyPolicy$new(epsilon = 0.1)
bandit      <- SyntheticBandit$new(weights = c(0.9, 0.1, 0.1))
agent       <- Agent$new(policy,bandit)
simulator   <- Simulator$new(agents = agent, simulations = 100, horizon = 80)
history     <- simulator$run()
```

In these lines, we start out by instantiating the `Policy` subclass `EpsilonGreedyPolicy` (covered in section 4.3) as object `policy`, with its parameter `epsilon` set to `0.1`. Next, we instantiate the `Bandit` subclass `SyntheticBandit` as `bandit`, with three Bernoulli arms, each offering a reward of one with probability p, and otherwise an reward of zero. For the current simulation, the `bandit`'s probability of reward is set to respectively 0.9, 0.1 and 0.1 per arm through its `weight` parameter. We then assign both our `bandit` and our `policy` to `Agent` instance `agent`. This `agent` is then added to a `Simulator` that is set to one hundred `simulations`, each with a `horizon` of one hundred—that is, `simulator` runs one hundred `simulations`, each with a different random seed, for one hundred time steps `t`.

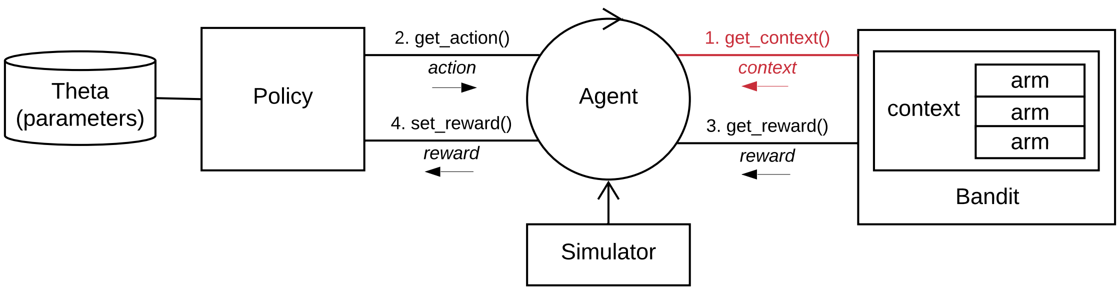


Figure 1: Diagram of **contextual**'s basic structure. The context feature vector returned by get_context() (colored red in the figure) is only taken into account by cMAB policies, and is ignored by MAB policies.

On running the `Simulator` it starts several (by default, the number of CPU cores minus one) worker processes, splitting simulations as efficiently as possible over each parallell worker. For each simulation, for every time step `t`, an `agent` clone then loops through each of the four function calls that constitute its main interior loop. Though we will delve deeper into the setup of each of **contextual**'s main classes in section 3, the current overview enables us to demonstrate how these four function calls relate to the four steps we defined in our cMAB formalization in section 2.1:

1) `agent` calls `bandit$get_context(t)`, which returns named list `list(k = n_arms, d = n_features, X = context)` containing the current `d` dimensional `context` feature vector `X` together with the number of arms `k`.

2) `agent` calls `policy$get_action(t, X)`, whereupon `policy` relays which arm to play based on the current `context` vector `X` (in MAB policies, X is ignored) and `theta` (the named list holding the parameters summarizing past `contexts`, `actions` and `rewards`). `policy` then returns a named list `list(choice = arm_chosen_by_policy)` that holds the index of the arm to play.

3) `agent` calls `bandit$get_reward(t, context, action)`, which returns a named list `list(reward = reward_for_choice_made, optimal_reward_value = optimal_reward_value)` that contains the `reward` for the `action` returned by `policy` in [2] and, optionally, the `optimal` reward at the current time `t` — if and when known.

4) `agent` calls `policy$set_reward(t, context, action, reward)` and uses the `action` taken, the `reward` received, and the current `context` to update the set of parameter values in `theta`

On completion of all of its simulation runs, `Simulator` returns an `history` object that contains a complete log of all interactions, which can, among others, be printed, plotted, or summarized:

```
summary(history)

Agents:

  EpsilonGreedy

Cumulative Regret:

         agent  t sims cum_regret cum_regret_var cum_regret_sd cum_regret_ci
 EpsilonGreedy 80  100      10.82       78.41172      8.855039      1.735556


Cumulative Reward:

         agent  t sims cum_reward cum_reward_var cum_reward_sd cum_reward_ci
 EpsilonGreedy 80  100      28.35       104.9975      10.24683      2.008341


Relative Cumulative Reward / Click Through Rate:

         agent  t sims ctr_reward ctr_reward_var ctr_reward_sd ctr_reward_ci
 EpsilonGreedy 80  100   0.354375       1.312468     0.1280853    0.02510427
```

3. R6 class structure

Since it is the **contextual**'s explicit goal to offer researchers and developers an easily extensible framework to develop, test and compare their own `Policy` and `Bandit` implementations, the current section offers additional background information on **contextual**' class structure—both on the R6 class system and on each of the six previously introduced core **contextual** classes.

3.1. R and the R6 Class System

Statistical computational methods, in R or otherwise, are regularly made available through single-use scripts or basic, isolated code packages. Usually, such code examples are meant to give a basic idea of a statistical method, technique or algorithm in the context of a scientific paper. Such code examples offer their scientific audience a rough inroad towards the comparison and further implementation of their underlying methods. However, when a set of well-researched interrelated algorithms, such as MAB and cMAB policies, find growing academic, practical and commercial adoption, it becomes crucial to offer a more standardized and more accessible way to compare such methods and algorithms.

It is against this background that we decided to develop the **contextual** R package—a package that would offer an easily extendible and open bandit framework together with extensible bandit and policy libraries. To us, it made the most sense to create such a package in R, as R is currently the de facto language for the dissemination of new statistical methods, techniques, and algorithms—while it is at the same time finding ever-growing adoption in industry. The resulting lively exchange of R related code, data, and knowledge between scientists and practitioners offers precisely the kind of cross-pollination that **contextual** hopes to facilitate.

Though widely used as a procedural language, R offers several Object Oriented (OO) systems, which can significantly help in structuring the development of more complex packages. Out of the OO systems available (S3, S4, R5 and R6), we settled on R6, as it offered several advantages compared to the other options. Firstly, it implements a mature object-oriented design when compared to S3. Secondly, its classes can be accessed and modified by reference—which offers the added advantage that R6 classes are instantly recognizable for developers with a background in Java or C++. Finally, when compared to the older R5 reference class system, R6 classes are lighter-weight and (as they do not make use of S4 classes) do not require the methods package—which makes **contextual** substantially less resource-hungry than it would otherwise have been.

3.2. Main classes

In this section, we go over each of **contextual**'s six main classes in some more detail—with an emphasis on the Bandit and Policy classes. To clarify **contextual**'s class structure, we also include two UML diagrams (UML or "Unified Modeling Language" is a modeling language that presents a standard way to visualize the overall class structure and general design of a software application or framework). The UML class diagram shown in Figure 10 on page 27 visualizes **contextual**'s static object model, showing how its classes inherit from, and interface with, each other. The UML sequence diagram in figure Figure 10 on page 28, on the other hand, illustrates how **contextual**'s classes interact dynamically over time.

Bandit

In **contextual**, any bandit implementation is expected to subclass and extend the Bandit superclass. It is then up to these subclasses themselves to provide an implementation for each of its abstract methods.`Bandit` subclasses are furthermore expected to set instance variable `self$k` to the number of arms, and `self$d` to the number of context features. On meeting this requirement, a `Bandit` is then required to implement `get_context()` and `do_action()`:

```
#' @export
Bandit <- R6::R6Class(
  portable = TRUE,
  class    = FALSE,
  public   = list(
    k            = NULL,      # Number of arms (integer)
    d            = NULL,      # Dimension of context feature vector (integer)
    ...
    precaching  = FALSE,     # Pregenerate context & reward matrices? (boolean)
    class_name  = "Bandit", # Bandit name - required (character)
    initialize  = function() {
      # Initialize Bandit. Generally, set self$d and self$k here.
    },
    ...
    get_context = function(t) {
      stop("Bandit subclass needs to implement bandit$get_context()", call. = FALSE)
      # Return a list with self$k, self$d and, where applicable, a context matrix X.
      list(X = context, k = arms, d = features)
    },
    get_reward = function(t, context, action) {
      stop("Bandit subclass needs to implement bandit$get_reward()", call. = FALSE)
      # Return a list with the reward and, if known, the reward of the best arm.
      list(reward = reward_for_choice_made, optimal = optimal_reward_value)
    },
    ...
  )
)
```

Bandit's functions can be described as follows:

- `new()` Generates and initializes a new `Bandit` object.
- `pre_calculate()` Called right after `Simulator` sets its seed, but before it starts iterating over all time steps `t` in T. If you need to initialize random values in a `Policy`, this is the place to do so.
- `get_context(t)` Returns a named list `list(k = n_arms, d = n_features, X = context)` with the current `d` dimensional `context` feature vector `X` together with the number of arms `k`.
- `get_reward(t, context, action)` Returns the named list `list(reward = reward_for_choice_made, optimal = optimal_reward_value)` containing the `reward` for the `action` previously returned by `policy` and, optionally, the `optimal` reward at the current time `t`.
- `generate_bandit_data()` A helper function that is called before `Simulator` starts iterating over all time steps `t` in T. This function is called when `bandit$precaching` has been set to `TRUE`. Pregenerate `contexts` and `rewards` here.

Where possible, it is advisable to pregenerate or precache `Bandit` contexts and rewards, as this is (as is generally the case in R) computationally much more efficient than the repeated generation of reward vectors and context matrices. This pregeneration can be implemented in `generate_bandit_data()`. It is called during a `Bandit`'s initialization—if and when the `Bandit`'s `self$precaching` variable is `TRUE`.

We also made several `Bandit` subclasses available. For each Bandit, there is at least one example script, to be found in the package's demo directory:

- `BasicBandit`: this basic (non-contextual) k-armed bandit synthetically generates rewards based on a weight vector. It returns a unit vector for context matrix X.
- `SyntheticBandit`: an example of a more complex and versatile synthetic bandit. It pregenerates both a randomized context matrix and reward vectors
- `ContextualBandit`: a contextual bandit that synthetically generates contextual rewards based on randomly set weights. It can simulate mixed user (cross-arm) and article (arm) feature vectors, generated from parameters k, d and num_users.
- `ContinuumBandit`: a basic example of a continuum bandit.
- `LiBandit`: a basic example of a bandit that makes use of offline data - here, an implementation of Li's [reference].

Each of these bandits can be used to run policies without further ado. They can, however, also be used as superclasses for custom `Bandit` subclas implementations. Or as templates for `Bandit` implementation(s) that directly subclass **contextual**'s `Bandit` superclass.

Policy

`Policy` is another central and often subclassed contexual superclass. Just like `Bandit`, this abstract class declares methods without itself offering an implementation. Any `Policy` subclass is therefore expected to implement `get_action()` and `set_reward()`. Also, any parameters that keep track or summarize `context`, `action` and `reward` values are required to be saved to `Policy`'s public named list `theta`.

```r
#' @export
Policy <- R6::R6Class(
  portable = FALSE,
  class = FALSE,
  public = list(
    action         = NULL,      # action results (list)
    theta          = NULL,      # policy parameters theta (list)
    theta_to_arms  = NULL,      # theta to arms "helper" (list)
    class_name     = "Policy",  # policy name - required (character)
    initialize = function() {
      self$theta  <- list()     # initializes theta list
      self$action <- list()     # initializes action list
    },
    get_action = function(t, context) {
      # Selects an arm based on self$theta and context, returns it in action$choice
```

```
      stop("Policy$get_action() has not been implemented.", call. = FALSE)
    },
    set_reward = function(t, context, action, reward) {
      # Updates parameters in theta based on reward awarded by bandit
      stop("Policy$set_reward() has not been implemented.", call. = FALSE)
    },
    set_parameters = function(context_params) {
      # Policy parameter (not theta!) initialisation happens here
      stop("Policy$set_parameters() has not been implemented.", call. = FALSE)
    },
    initialize_theta = function(k) {
      # Called during contextual's initialisation.
      # Copies theta_to_arms k times, makes the copies available through theta.
      ...
    }
  )
)
```

Bandit's functions can be described as following:

- `set_parameters()` This helper function, called during a Policy's initialisation, assigns the values it finds in list `self$theta_to_arms` to each of the Policy's k arms. The parameters defined here can then be accessed by arm index in the following way: `theta[[index_of_arm]]$parameter_name`.
- `get_action(t, context)` Calculates which arm to play based on the current values in named list `theta` and the current `context`. Returns a named list `list(choice = arm_chosen_by_policy)` that holds the index of the arm to play.
- `set_reward(t, context, action, reward)` Returns the named list `list(reward = reward_for_choice_made, optimal = optimal_reward_value)` containing the `reward` for the `action` previously returned by `policy` and, optionally, the `optimal` reward at the current time `t`.

Agent

To ease the encapsulation of parallel `Bandit` and `Policy` simulations, `Agent` is responsibe for the flow of information between and the running of one `Bandit` and `Policy` pair, for example:

```
policy              <- EpsilonGreedyPolicy$new(epsilon = 0.1, name = "EG")
bandit              <- SyntheticBandit$new(weights = c(0.9, 0.1, 0.1))
agent               <- Agent$new(policy,bandit)
```

It does this by keeping track of `t` through its private named list variable `state` and by making sure that, at each time step `t`, all four main `Bandit` and `Policy` cMAB methods are called in correct order:

```
Agent <- R6::R6Class(
  public = list(
    #...
```

```
    do_step = function() {
      t <- t + 1
      context = bandit$get_context(t)
      action  = policy$get_action (t, context)
      reward  = bandit$get_reward (t, context, action)
      theta   = policy$set_reward (t, context, action, reward)
      list(context = context, action = action, reward = reward,theta = theta)
    }
    #...
  )
)
```

Its main function is `do_step()`, generally called by the worker of a `Simulator` object that takes care of the running of a particular `agent` instance:

- `do_step()` Completes one time step `t` by consecutively calling `bandit$get_context()`, `policy$get_action()`, `bandit$get_reward()` and `policy$set_reward()`.

Simulator

A Simulator instance is the entry point of any **contextual** simulation. It encapsulates one or more `Agents`, clones them if necessary, runs the `Agents` (in parallel, by default), and saves the log of all of the `Agents` interactions to a `History` object:

```
history <- Simulator$new(agents = agent, horizon = 10, simulations = 10)$run()
```

By default, for performance reasons, a `Simulator` does not save `context` matrices and the (potentially deeply nested) `theta` list to its `History log`. But with its `save_context` and `save_theta` arguments set to `TRUE`, a the above history data.table object would be structured as follows:

```
print(history)

 $ t                   : int  1 2 3 4 5 6 7 8 9 10 ...  # time step
 $ sim                 : int  1 1 1 1 1 1 1 1 1 1 ...   # simulation index number
 $ choice              : num  3 1 1 1 1 1 1 1 1 1 ...   # arm chosen by policy
 $ reward              : num  0 1 0 1 1 1 1 1 1 1 ...   # reward awarded by bandit
 $ choice_is_optimal   : int  0 1 1 1 1 1 1 1 1 1 ...   # chosen arm optimal one?
 $ optimal_reward_value: num  1 1 0 1 1 1 1 1 1 1 ...   # best reward at t
 $ propensity          : num  0.9 0.9 0.033 0.9 0.9 ... # propensity of chosen arm
 $ agent               : chr  'ThompsonSampling' ...    i# agent name
 $ context             : List of 100 matrices           # list of context matrices
 $ theta               : List of 100 Lists              # list of (nested) thetas
```

To specify how to run a simulation and which data is to be saved to a `Simulator` instance's `History` log, a Simulator object can be configured through the following parameters:

- `agents` An `Agent` instance, or a `list` of `Agent` instances to be run by the instantiated `Simulator`.

- `horizon` The T time steps to run the instantiated `Simulator`.
- `simulations` How many times to repeat each agent's simulation with a new seed on each repeat (itself deterministically derived from set_seed).
- `save_context` Save context matrices `X` to the History log during a simulation?
- `save_theta` Save the parameter list `theta` to the History log during a simulation?
- `do_parallel` Run `Simulator` processes in parallel?
- `worker_max` Specifies how many parallel workers are to be used, when `do_parallel` is `TRUE`. If unspecified, the amount of workers defaults to `max(workers_available)-1`.
- `continuous_counter` Of use to, among others, offline Bandits. If `continuous_counter` is set to `TRUE`, the current `Simulator` iterates over all rows in a data set for each repeated simulation. If `FALSE`, it splits the data into `simulations` parts, and a different subset of the data for each repeat of an agent's simulation.
- `set_seed` Sets the seed of R's random number generator for the current `Simulator`.
- `write_progress_file` If `TRUE`, `Simulator` writes `progress.log` and `doparallel.log` files to the current working directory, allowing you to keep track of `workers`, iterations, and potential errors when running a `Simulator` in parallel.
- `include_packages` List of packages that (one of) the policies depend on. If a `Policy` requires an R package to be loaded, this option can be used to load that package on each of the workers. Ignored if `do_parallel` is `FALSE`.
- `reindex_t` If `TRUE`, removes empty rows from the `History` log, re-indexes the `t` column, and truncates the resulting data to the shortest simulation grouped by agent and simulation.

History

A `Simulator` aggregates the data acquired during a simulation in a `History` object's private `data.table` log. It also calculates per agent average cumulative reward, and, when the optimal outcome per `t` is known, per agent average cumulative regret. It is furthermore possible to `plot()` a `History object`, `summarize()` it, and, among others, obtain either a `data.frame()` or a `data.table()` from any `History` instance:

```
history            <- Simulator$new(agent)$run()
dt                 <- history$get_data_table()
df                 <- history$get_data_frame()
cumulative_regret  <- history$cumulative(regret = TRUE)
```

Some other History functions:

- `save(index, t, action, reward, policy_name, simulation_index, context_value = NA, theta_value = NA)` Saves one row of simulation data. `save()` is generally not called directly, but trough a `Simulator` instance.

- `save_data(filename = NA)` Writes the `History` log file in its default data.table format, with `filename` as the name of the file which the data is to be written to.
- `load_data = function(filename, nth_rows = 0)` Reads a `History` log file in its default `data.table` format, with `filename` as the name of the file which the data are to be read from. If `nth_rows` is larger than 0, every `nth_rows` of data is read instead of the full data file. This can be of use with (a first) analysis of very large data files.
- `reindex_t(truncate = TRUE)` Removes empty rows from the `History` log, reindexes the `t` column, and, if `truncate` is TRUE, truncates the resulting data to the shortest simulation grouped by agent and simulation.
- `print_data()` Prints a summary of the `History` log.
- `cumulative(final = TRUE, regret = TRUE, rate = FALSE)` Returns cumulative reward (when `regret` is FALSE) or regret. When `final` is TRUE, it only returns the final value. When `final` is FALSE, it returns a `data.table` containing all cumulative reward or regret values from 1 to T. When `rate` is TRUE, cumulative reward or regret are divided by column `t` before any values are returned.

Plot

The `Plot` class takes an `History` object, and offers several default types of plot:

- `average`: plots the average reward or regret over all simulations per Agent (that is, each Bandit and Policy combo) over time.
- `cumulative`: plots the average reward or regret over all simulations per Agent over time.
- `optimal`: if data on optimal choice is available, "optimal" plots how often the best or optimal arm was chosen on average at each timestep, in percentages, over all simulations per Agent.
- `arms`: plots ratio of arms chosen on average at each time step, in percentages, totaling 100

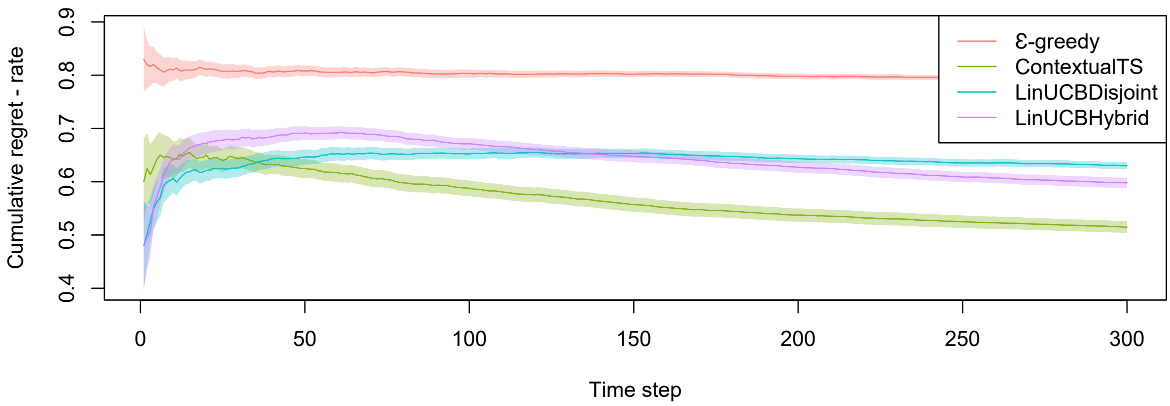
`Plot` objects can be instantiated directly, or, more commonly, by calling the `plot()` function. In either case, make sure to specify a `History` instance and one of the plot types specified above:

```
# plot a history object through default generic plot() function
plot(history, type = "arms")

# or use the Plot class directly
p1 <- Plot$new()$cumulative(history)
p2 <- Plot$new()$average(history)
```

Also, multiple agents can be combined into one plot:

```
bandit  <- ContextualBandit$new(k = 5, d = 6, num_users = 7)
agents  <- list(
  Agent$new(EpsilonGreedyPolicy$new(0.1, "\U190-greedy"), bandit),
```

```
  Agent$new(ContextualThompsonSamplingPolicy$new(name = "ContextualTS"), bandit),
  Agent$new(LinUCBDisjointPolicy$new(0.7, "LinUCBDisjoint"), bandit),
  Agent$new(LinUCBHybridPolicy$new(0.7, 6, "LinUCBHybrid"), bandit)
)
history <- Simulator$new(agents, 300, 100)$run()

# plot using contextual - faster, less code and configuration
plot(history, type = "cumulative", ci = TRUE, rate = TRUE)
```



4. Implementing and extending Policy and Bandit subclasses

Though section 3 provides an introduction to all of **contextual**'s main classes, in practice, researchers will mostly focus on subclassing `Policies` and `Bandits`. The current section therefore first demonstrates how to implement some well-known bandit algorithms, and, secondly, how to create `Policy` and `Bandit` sub-subclasses.

4.1. BasicBandit: a Minimal Bernoulli Bandit

Where not otherwise noted, all `Bandit` implementations in the current paper refer to (or will be configured as) multi-armed `Bandits` with Bernoulli rewards. For Bernoulli `Bandits`, the reward received is either a zero or a one: on each t they offer either a reward of 1 with probability p or a reward of 0 with probability $p - 1$. In other words, a Bernoulli Bandit has a finite set of arms a $\in \{1, \ldots, k\}$ where the rewards for each arm a is distributed Bernoulli with parameter μa, the expected reward of the arm.

One example of a very simple non-contextual Bernoulli bandit is **contextual**'s minimal `Bandit` implementation, `BasicBandit`:

```
BasicBandit <- R6::R6Class(
  inherit = Bandit,
  public = list(
    initialize    = function(weights = NULL) {
      self$set_weights(weights)                          # arm weight vector
      private$X <- array(1, dim = c(self$d, self$k, 1)) # context matrix of ones
    },
    # ...
```

```
    get_weights = function() {
      private$W
    },
    set_weights = function(local_W) {
      private$W <- matrix(local_W, nrow = 1L)           # arm weight vector
      self$d    <- as.integer(dim(private$W)[1])        # context dimensions
      self$k    <- as.integer(dim(private$W)[2])        # arms
      private$W
    },
    get_context = function(t) {
      contextlist <- list(k = self$k, d = self$d, X = private$X)
      contextlist
    },
    get_reward = function(t, context, action) {
      private$R  <- as.double(matrix(runif(self$k) < get_weights(), self$k, self$d))
      rewardlist <- list(
        reward                   = private$R[action$choice],
        optimal_reward_value     = private$R[which.max(private$R)]
      )
      rewardlist
    }
    # ...
  )
)
```

`BasicBandit` expects a `weight` vector of probabilities, where every element in `weight` represents the probability of `BasicBandit` returning a reward of 1 for one of its `k` arms. Also, observe that, at every `t`, `BasicBandit` sets private `context` matrix `X` to an unchanging, neutral `d` features times `k` arms unit matrix, which alludes to the fact that `BasicBandit` does not generate any covaried contextual cues related to its arms.

4.2. Epsilon First

An important feature of **contextual** is that it eases the conversion from formal and pseudocode policy descriptions to clean R6 classes. We will give several examples of such conversions in the current paper, starting with the implementation of the Epsilon First algorithm. In this non-contextual algorithm, also known as AB(C) testing, a pure exploration phase is followed by a pure exploitation phase.

In that respect, the Epsilon First algorithm is actually equivalent to a randomized controlled trial (RCT). An RCT, generally refered to as the gold standard clinical research paradigm, is a study design where subjects are allocated at random to receive one of several clinical interventions. On completion of an RCT, the most succesful intervention up till that point in time is suggested to be the superior "evidence-based" option from then on.

A more formal pseudocode description of this Epsilon First policy:

Algorithm 1 Epsilon First

Require: $\eta \in \mathbb{Z}^+$, number of time steps t in the exploration phase
  $n_a \leftarrow 0$ for all arms a $\in \{1, \ldots, k\}$ (count how many times an arm has been chosen)
  $\hat{\mu}_a \leftarrow 0$ for all arms a $\in \{1, \ldots, k\}$ (estimate of expected reward per arm)
  for $t = 1, \ldots, T$ **do**
    if $t \leq \eta$ **then**
      play a random arm out of all arms a $\in \{1, \ldots, k\}$
    else
      play arm $a_t = \arg\max_a \hat{\mu}_{t=\eta,a}$ with ties broken arbitrarily
    end if
    observe real-valued payoff r_t
    $n_{a_t} \leftarrow n_{a_{t-1}} + 1$
    $\hat{\mu}_{t,a_t} \leftarrow \dfrac{r_t - \hat{\mu}_{t-1,a_t}}{n_{a_t}}$
  end for

And the above pseudocode converted to an `EpsilonFirstPolicy` class:

```
EpsilonFirstPolicy <- R6::R6Class(
  public = list(
    first = NULL,
    initialize = function(first = 100) {
      super$initialize(name)
      self$first <- first
    },
    set_parameters = function() {
      self$theta_to_arms <- list('n' = 0, 'mean' = 0)
    },
    get_action = function(context, t) {
      if (sum_of(theta$n) < first) {
        action$choice          <- sample.int(context$k, 1, replace = TRUE)
        action$propensity   <- (1/context$k)
      } else {
        action$choice          <- max_in(theta$mean, equal_is_random = FALSE)
        action$propensity   <- 1
      }
      action
    },
    set_reward = function(context, action, reward, t) {
      arm      <- action$choice
      reward   <- reward$reward

      inc(theta$n[[arm]]) <- 1
      if (sum_of(theta$n) < first - 1)
        inc(theta$mean[[arm]] ) <- (reward - theta$mean[[arm]]) / theta$n[[arm]]

      theta
    }
  )
)
```

To evaluate this policy, instantiate both an `EpsilonFirstPolicy` and a `SyntheticBandit` (a contextual and more versatile `BasicBandit` subclass). Then add the `Bandit/Policy` pair to an `Agent`. Next, add the `Agent` to a `Simulator`. Finally, run the `Simulator`, and `plot()` the its `History` log:

```
horizon            <- 100
simulations        <- 1000
weights            <- c(0.6, 0.3, 0.3)

policy             <- EpsilonFirstPolicy$new(first = 50)
bandit             <- SyntheticBandit$new(weights = weights)

agent              <- Agent$new(policy,bandit)

simulator          <- Simulator$new(agents = agent,
                                    horizon = horizon,
                                    simulations = simulations,
                                    do_parallel = FALSE)

history            <- simulator$run()

par(mfrow = c(1, 2), mar = c(2,4,1,1))
plot(history, type = "cumulative", no_par = TRUE)
plot(history, type = "arms", no_par = TRUE)
```

4.3. The Epsilon Greedy Policy

Contrary to the previously introduced Epsilon First policy, an Epsilon Greedy algorithm does not divide exploitation and exploration into two strictly separate phases—it explores with a probability of $epsilon$ and exploits with a probability of $1 - epsilon$, right from the start. That is, an Epsilon Greedy policy with an $epsilon$ of 0.1 explores arms at random 10% of the time. The other $1 - epsilon$, or 90% of the time, the policy "greedily" exploits the currently best-known arm.

This can be formalized in pseudocode as follows:

Algorithm 2 Epsilon Greedy

Require: $\epsilon \in [0, 1]$ - exploration tuning parameter
  $n_a \leftarrow 0$ for all arms a $\in \{1, \ldots, k\}$ (count how many times an arm has been chosen)
  $\hat{\mu}_a \leftarrow 0$ for all arms a $\in \{1, \ldots, k\}$ (estimate of expected reward per arm)
  for $t = 1, \ldots, T$ **do**
    if sample from $\mathcal{N}(0, 1) > \epsilon$ **then**
      play arm $a_t = \arg\max_a \hat{\mu}_{t-1,a}$ with ties broken arbitrarily
    else
      play a random arm out of all arms a $\in \{1, \ldots, k\}$
    end if
    observe real-valued payoff r_t
    $n_{a_t} \leftarrow n_{a_{t-1}} + 1$
    $\hat{\mu}_{t,a_t} \leftarrow \dfrac{r_t - \hat{\mu}_{t-1,a_t}}{n_{a_t}}$
  end for

Converted to an EpsilonGreedyPolicy class:

```
EpsilonGreedyPolicy <- R6::R6Class(
  public = list(
    epsilon = NULL,
    initialize = function(epsilon = 0.1) {
      super$initialize(name)
      self$epsilon <- epsilon
    },
    set_parameters = function() {
      self$theta_to_arms <- list('n' = 0, 'mean' = 0)
    },
    get_action = function(context, t) {
      if (runif(1) > epsilon) {
        action$choice <- max_in(theta$mean)
        action$propensity <- 1 - self$epsilon
      } else {
        action$choice <- sample.int(context$k, 1, replace = TRUE)
        action$propensity <- epsilon*(1/context$k)
      }
      action
    },
    set_reward = function(context, action, reward, t) {
      arm <- action$choice
      reward <- reward$reward
      inc(theta$n[[arm]])    <- 1
      inc(theta$mean[[arm]]) <- (reward - theta$mean[[arm]]) / theta$n[[arm]]
      theta
    }
  )
)
```

Assign the new class, together with `SyntheticBandit`, to an `Agent`. Again, assign the `Agent` to a `Simulator`. Then run the `Simulator` and `plot()`:

```
horizon           <- 100
simulations       <- 1000
weights           <- c(0.6, 0.3, 0.3)

policy            <- EpsilonGreedyPolicy$new(epsilon = 0.1)
bandit            <- SyntheticBandit$new(weights = weights)

agent             <- Agent$new(policy,bandit)

simulator         <- Simulator$new(agents = agent,
                                   horizon = horizon,
                                   simulations = simulations,
                                   do_parallel = FALSE)

history           <- simulator$run()

par(mfrow = c(1, 2), mar = c(2,4,1,1))
plot(history, type = "cumulative", no_par = TRUE)
plot(history, type = "arms", no_par = TRUE)
```

4.4. Contextual Bandit: LinUCB with Linear Disjoint Models

As a final example of how to subclass **contextual**'s `Bandit` superclass, we move from non-contextual algorithms to a contextual one. As described in section 1, contextual bandits can make use of side information to help them choose the current best arm to play. For example, contextual information such as a website visitors' location may be related to which article's headline (or arm) on the frontpage of the website will be clicked on most.

Here, we show how to implement and evaluate probably one of the most cited out of all contextual policies, the LinUCB algorithm with Linear Disjoint Models. The policy is more complicated than the previous two bandits, but when following its pseudocode description to the letter, it translates nicely to yet another Bandit subclass.

The LinUCBDisjoint algorithm works by running a linear regression with coefficients for each of `d` contextual features on the available historical data. Then the algorithm observes the new context and uses this context to generate a predicted reward based on the regression model. Importantly, the algorithm also generates a confidence interval for the predicted payoff for each of `k` arms. The policy then chooses the arm with the highest upper confidence bound. In pseudocode:

Algorithm 3 LinUCB with linear disjoint models

Require: $\alpha \in \mathbb{R}^+$, exploration tuning parameter
  for $t = 1, \ldots, T$ **do**
    Observe features of all arms $a \in \mathcal{A}_t : x_{t,a} \in \mathbb{R}^d$
    for $a \in \mathcal{A}_t$ **do**
      if a is new **then**
        $A_a \leftarrow I_d$ (d-dimensional identity matrix)
        $b_a \leftarrow 0_{d\times 1}$ (d-dimensional zero vector)
      end if
      $\hat{\theta}_a \leftarrow A_a^{-1} b_a$
      $p_{t,a} \leftarrow \hat{\theta}_a^T + \alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t,a}}$
    end for
    Play arm $a_t = \arg\max_a p_{t,a}$ with ties broken arbitrarily and observe real-valued payoff r_t
    $A_{a_t} \leftarrow A_{a_t} + x_{t,a_t} x_{t,a_t}^T$
    $b_{a_t} \leftarrow b_{a_t} + r_t x_{t,a_t}$
  end for

Next, translating the above pseudocode into a well organized `Bandit` subclass:

```
#' @export
LinUCBDisjointPolicy <- R6::R6Class(
  public = list(
    alpha = NULL,
    initialize = function(alpha = 1.0) {
      super$initialize(name)
      self$alpha <- alpha
    },
    set_parameters = function() {
      self$theta_to_arms <- list( 'A' = diag(1,self$d,self$d), 'b' = rep(0,self$d))
    },
    get_action = function(context, t) {
      expected_rewards <- rep(0.0, context$k)
      for (arm in 1:self$k) {
        X          <-  context$X[,arm]
        A          <-  theta$A[[arm]]
        b          <-  theta$b[[arm]]
        A_inv      <-  solve(A)

        theta_hat  <-  A_inv %*% b
        mean       <-  X %*% theta_hat
        sd         <-  sqrt(tcrossprod(X %*% A_inv, X))
        expected_rewards[arm] <- mean + alpha * sd
      }
      action$choice  <- max_in(expected_rewards)
      action
    },
    set_reward = function(context, action, reward, t) {
      arm <- action$choice
      reward <- reward$reward
      Xa <- context$X[,arm]
```

```
      inc(theta$A[[arm]]) <- outer(Xa, Xa)
      inc(theta$b[[arm]]) <- reward * Xa

      theta
    }
  )
)
```

Now it is possible to evaluate the `LinUCBDisjointPolicy` using a Bernoulli `SyntheticBandit` with three arms and three context features. In the code below we define each of `SyntheticBandit`'s arms to be, on average, equally probable to return a reward. However, at the same time, the presence of a random context feature vector exercises a strong influence on the distribution of rewards over the arms per time step t: in the presence of a specific feature, one of the arms becomes much more likely to offer a reward. In this setting, the `EpsilonGreedyPolicy` does not do better than chance. But the `LinUCBDisjointPolicy` is able to learn the relationships between arms, rewards, and features without much difficulty:

```
horizon       <- 100L
simulations   <- 300L

                      # k=1  k=2  k=3              -> columns represent arms

weights      <- matrix(c(0.8, 0.1, 0.1,      # d=1   -> rows represent
                        0.1, 0.8, 0.1,      # d=2      context features
                        0.1, 0.1, 0.8),     # d=3

                        nrow = 3, ncol = 3, byrow = TRUE)

bandit       <- SyntheticBandit$new(weights = weights, precaching = TRUE)

agents       <- list(Agent$new(EpsilonGreedyPolicy$new(0.1), bandit, "EGreedy"),
                    Agent$new(LinUCBDisjointPolicy$new(1.0), bandit, "LinUCB"))

simulation   <- Simulator$new(agents, horizon, simulations, do_parallel = FALSE)
history      <- simulation$run()

par(mfrow = c(1, 2), mar = c(2,4,1,1))
plot(history, type = "cumulative", regret = FALSE, no_par = TRUE)
plot(history, type = "cumulative", no_par = TRUE)
```

4.5. Subclassing Policies and Bandits

contextual's extensibility does, of course, not limit itself to the subclassing of `Policy` classes. Through its R6 based object system it is easy to extend and override any **contextual** super- or subclass. Below, we demonstrate how to apply that extensibility to sub-subclass one Bandit and one Policy subclass. First, we extend `BasicBandit`'s codePoissonRewardBandit, replacing

`BasicBandit`'s Bernoulli based reward function with a Poisson based one. Next, we implement an `EpsilonGreedyAnnealingPolicy` version of the Epsilon Greedy policy introduced in section 4.2—where its `EpsilonGreedyAnnealingPolicy` subclass introduces a gradual reduction ("annealing") of the policy's *epsilon* parameter over T, in effect moving the policy from more explorative to a more exploitative over time.

```r
PoissonRewardBandit <- R6::R6Class(
  # Class extends BasicBandit
  inherit = BasicBandit,
  public = list(
    class_name = "PoissonRewardBandit",
    initialize = function(weights) {
      super$initialize(weights)
    },
    # Overrides BasicBandit's get_reward to generate Poisson based rewards
    get_reward = function(t, context, action) {
      reward_means = c(2,2,2)
      rpm <- rpois(3, reward_means)
      private$R <- matrix(rpm < self$get_weights(), self$k, self$d)*1
      list(
        reward                    = private$R[action$choice],
        optimal_reward_value      = private$R[which.max(private$R)]
      )
    }
  )
)
EpsilonGreedyAnnealingPolicy <- R6::R6Class(
  # Class extends EpsilonGreedyPolicy
  inherit = EpsilonGreedyPolicy,
  portable = FALSE,
  public = list(
    class_name = "EpsilonGreedyAnnealingPolicy",
    # Override EpsilonGreedyPolicy's get_action, use annealing epsilon
    get_action = function(t, context) {
      self$epsilon <- 1 / log(t + 0.0000001)
      super$get_action(t, context)
    }
  )
)
weights     <- c(7,1,2)
horizon     <- 200
simulations <- 100
bandit      <- PoissonRewardBandit$new(weights)
agents      <- list(Agent$new(EpsilonGreedyPolicy$new(0.1), bandit, "EG Annealing"),
                    Agent$new(EpsilonGreedyAnnealingPolicy$new(0.1), bandit, "EG"))
simulation  <- Simulator$new(agents, horizon, simulations, do_parallel = FALSE)

history     <- simulation$run()

par(mfrow = c(1, 2), mar = c(2,4,1,1))
plot(history, type = "cumulative", no_par = TRUE)
plot(history, type = "average", regret = FALSE, no_par = TRUE)
```

5. Offline evaluation

Though it is, as demonstrated in the previous section, relatively easy to create basic synthetic Bandits to test simple MAB and cMAB policies, the creation of more elaborate simulations that generate more complex contexts for more demanding policies can become very complicated very fast. So much so, that the implementation of such simulators regularly becomes more intricate than the analysis and implementation of the policies themselves. Moreover, even when succeeding in surpassing these technical challenges, it remains an open question if an evaluation based on simulated data reflects real-world applications since modeling by definition introduces bias.

It would, of course, be possible to evaluate policies by running them in a live setting. Such live evaluations would undoubtedly deliver unbiased, realistic estimates of a policy's effectiveness. However, the use of live data makes it more difficult to compare multiple policies at the same, as it is not possible to test multiple policies at the same time with the same user. Using live data is usually also much slower than an offline evaluation, as online evaluations are dependent on active user interventions. Furthermore, the testing of policies on a live target audience, such as patients or customers, with potentially suboptimal policies, could become either dangerous or very expensive.

Another unbiased approach to testing MAB and cMAB policies would be to make use of offline historical data or logs. Such a data source does need to contain observed contexts and rewards, and any actions or arms must have been selected either at random or with a known probability per arm $D = (p_1, p_2, p_3, ..., p_k)$. That is, such data sets contain at least $D = (x_{t,a_t}, a_t, r_{t,a_t})$, or, in the case of know probabilities per arm $D = (x_{t,a_t}, a_t, r_{t,a_t}, p_a)$. Not only does such offline data pre-empt the issues of bias and model complexity, but it also offers the advantage that such data is widely available, as historical logs, as benchmark data sets for supervised learning, and more.

There is a catch though; when we make use of offline data, we miss out on user feedback every time a policy under evaluation suggests a different arm from the one that was initially selected and saved to the offline data set. In other words, offline data is "partially labeled" in respect to evaluated Bandit policies. However, as shown in the following subsections, it is possible to get around this partial labeling problem by discarding part of the data, and by making the most of any additional information in offline data sets.

5.1. Offline Evaluation of Policies through LiSamplingBandit

The first, and most important, step in using offline data in policy evaluation is to recognize that we need to limit our evaluation to those rows of data where the arm selected is the same as the one that is suggested by the policy under evaluation. In pseudocode:

Algorithm 4 Li Policy Evaluator

Require: Policy π

Data stream of events S of length T

$h_0 \leftarrow \emptyset$ An initially empty history log

$R_\pi \leftarrow 0$ An initially zero total cumulative reward

$L \leftarrow 0$ An initially zero length counter of valid events

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

  Get the t-th event $(x_{t,a_t}, a_t, r_{t,a_t})$ from S

  if $\pi(h_{t-1}, x_{t,a_t}) = a_t$ **then**

    $h_t \leftarrow \text{CONCATENATE}(h_{t-1}, (x_{t,a_t}, a_t, r_{t,a_t}))$

    $R_\pi = R_\pi + r_{t,a_t}$

    $L = L + 1$

  else

    $h_t \leftarrow h_{t-1}$

  end if

end for

Output: rate of cumulative regret R_π / L

Converted to a contextual BasicBandit subclass and run on a large data set:

```
BasicLiBandit <- R6::R6Class(
  "BasicLiBandit",
  inherit = BasicBandit,
  portable = TRUE,
  class = FALSE,
  private = list(
    S = NULL
  ),
  public = list(
    initialize   = function(data_stream, arms) {
      self$k <- arms
      self$d <- 1
      private$S <- data_stream
    },
    get_context = function(index) {
      contextlist <- list(
        k = self$k,
        d = self$d,
        X = matrix(1,self$d,self$k) # no context
      )
      contextlist
    },
    get_reward = function(index, context, action) {
      reward_at_index <- as.double(private$S$reward[[index]])
      if (private$S$choice[[index]] == action$choice) {
        list(reward = reward_at_index)
      } else {
        NULL
      }
    }
```

```
  )
)

library(data.table)
library(RCurl)
library(foreign)

url <- "https://raw.githubusercontent.com/Nth-iteration-labs/"
url <- paste0(url,"contextual_data/master/PersuasionAPI/persuasion.csv")

website_data    <- getURL(url)
website_data    <- setDT(read.csv(textConnection(persuasion_data)))
horizon         <- 350000L
simulations     <- 100L
bandit          <- BasicLiBandit$new(website_data, arms = 4)
agent           <- Agent$new(UCB1Policy$new(), bandit)
history         <- Simulator$new(agent, horizon, simulations, reindex_t = TRUE)$run()

par(mfrow = c(1, 2), mar = c(2,4,1,1))
plot(history, type = "cumulative", regret = FALSE,
     rate = TRUE, ylim = c(0.0105, 0.015))
```

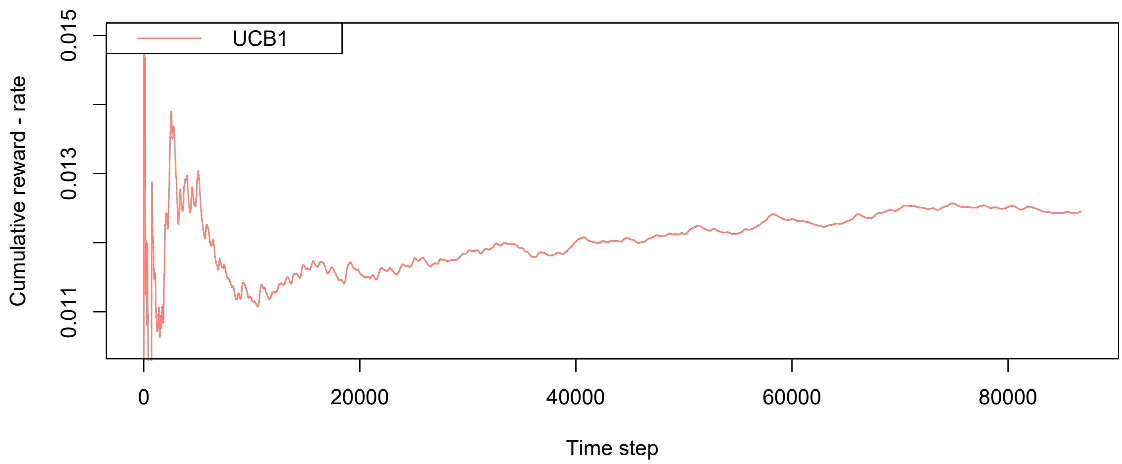


Figure 2: An UCB1 policy evaluated on a Li Bandit. The Bandit samples from 350000 rows with clicks for rewards and the display of one of four “persuasive strategies” to users of an online store representing the offline Bandit’s four arms.

Offline evaluation through DoublyRobustBandit

*** insert algorithm *** *** insert code ***

6. Replication of existing studies

Here replication with yahoo dataset

7. Parallelization

General

Parallel is great!

Amazon EC2

Amazon EC2 rocks.

Microsoft Azure

Azure is cool too.

Tradeoffs

Time is of the essence...

8. Extra greedy UCB

Now with extra greedy!
And lock in feedback too? And dueling? and..

9. Conclusion

10. UML Diagrams

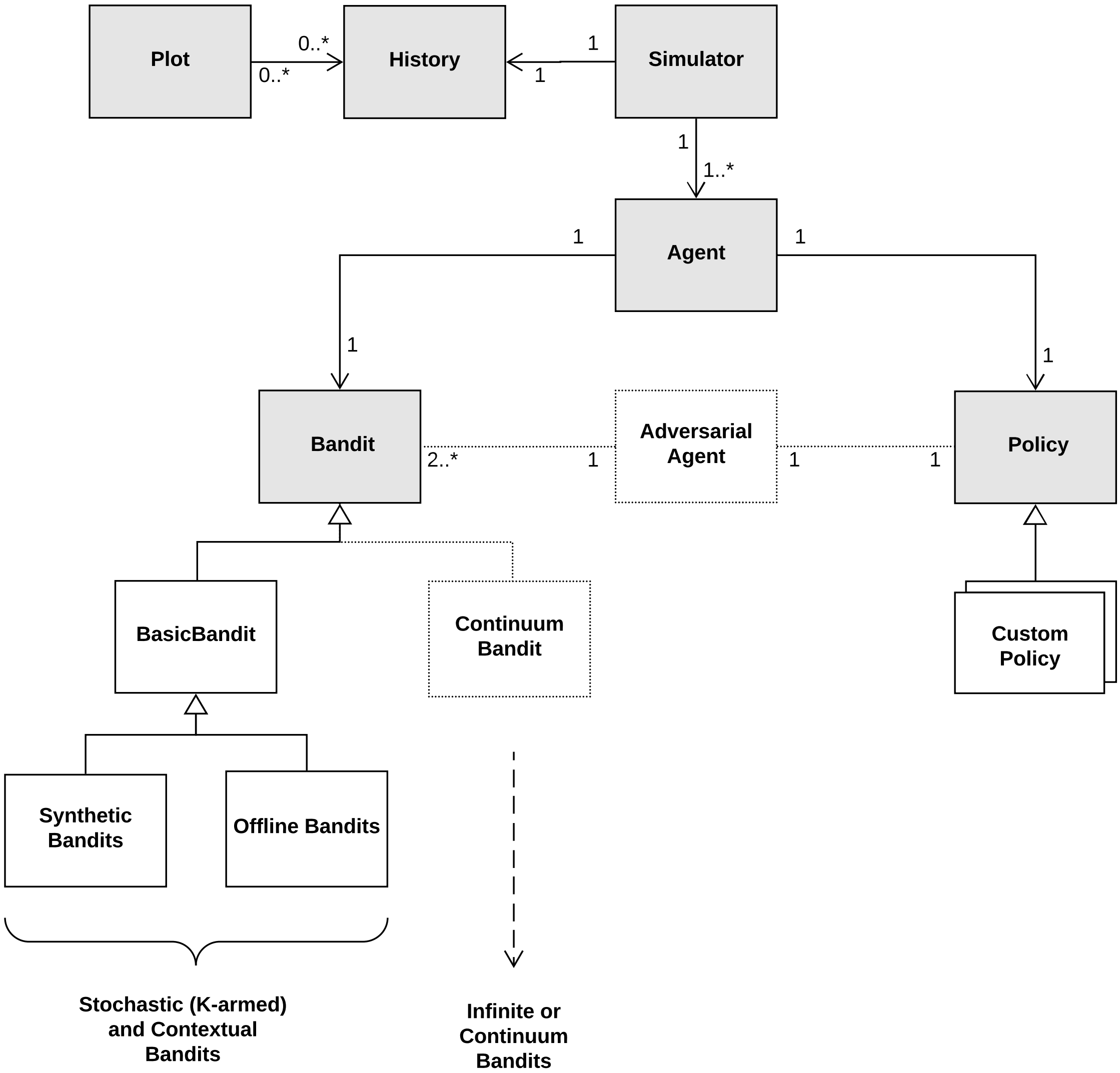


Figure 3: **contextual** UML Class Diagram

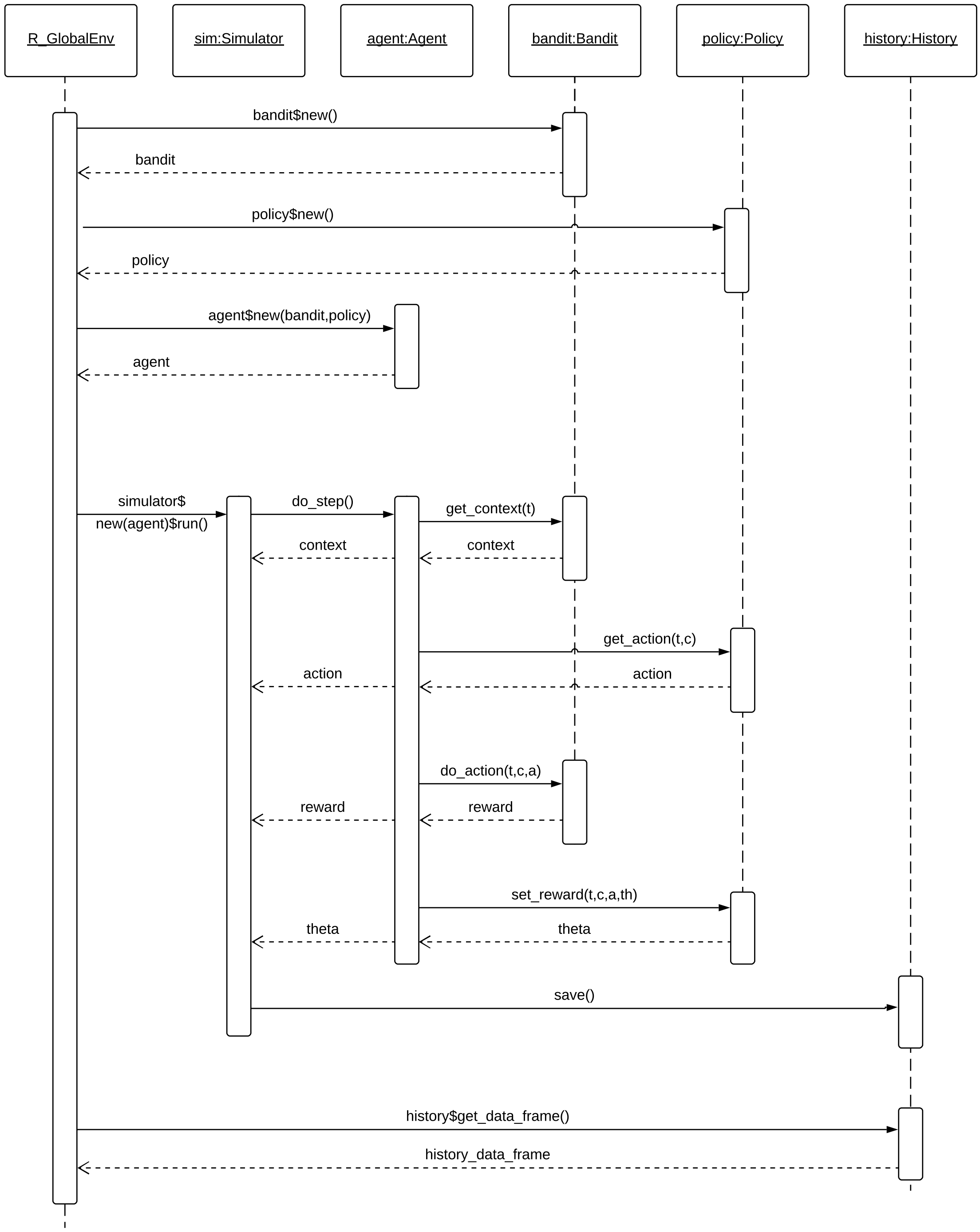


Figure 4: **contextual** UML Sequence Diagram

11. Acknowledgments

Thanks go to my colleagues at the Jheronimus Academy of Data Science and Tilburg University!

References

(2018). "striatum: Contextual bandit in python." Original-date: 2016-05-11T06:26:38Z, URL https://github.com/ntucllab/striatum.

Abe N, Biermann AW, Long PM (2003). "Reinforcement learning with immediate rewards and linear hypotheses." *Algorithmica*, **37**(4), 263–293.

Auer P, Cesa-Bianchi N, Fischer P (2002). "Finite-time analysis of the multiarmed bandit problem." *Machine learning*, **47**(2-3), 235–256.

Besson L (2018). "SMPyBandits: an Open-Source Research Framework for Single and Multi-Players Multi-Arms Bandits (MAB) Algorithms in Python." Online at: github.com/SMPyBandits/SMPyBandits. Code at https://github.com/SMPyBandits/SMPyBandits/, documentation at https://smpybandits.github.io/, URL https://github.com/SMPyBandits/SMPyBandits/.

Bubeck S, Cesa-Bianchi N, *et al.* (2012). "Regret analysis of stochastic and nonstochastic multi-armed bandit problems." *Foundations and Trends® in Machine Learning*, **5**(1), 1–122.

Kaelbling LP, Littman ML, Moore AW (1996). "Reinforcement learning: A survey." *Journal of artificial intelligence research*, **4**, 237–285.

Kaptein MC, Kruijswijk JMA (2016). "Streamingbandit: A platform for developing adaptive persuasive systems." *JSS*.

Lai TL, Robbins H (1985). "Asymptotically efficient adaptive allocation rules." *Advances in applied mathematics*, **6**(1), 4–22.

Langford J, Li L, Strehl A (2007). "Vowpal {W}abbit."

Langford J, Zhang T (2008). "The epoch-greedy algorithm for multi-armed bandits with side information." In *Advances in neural information processing systems*, pp. 817–824.

Lu T, Pál D, Pál M (2010). "Contextual multi-armed bandits." In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics*, pp. 485–492.

R Core Team (????). "R: A Language and Environment for Statistical Computing." URL https://www.R-project.org.

Sarkar J (1991). "One-armed bandit problems with covariates." *The Annals of Statistics*, pp. 1978–2002.

Strehl AL, Mesterharm C, Littman ML, Hirsh H (2006). "Experience-efficient learning in associative bandit problems." In *Proceedings of the 23rd international conference on Machine learning*, pp. 889–896. ACM.

Tang L, Rosales R, Singh A, Agarwal D (2013). "Automatic ad format selection via contextual bandits." In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pp. 1587–1594. ACM.

Tewari A, Murphy SA (2017). "From ads to interventions: Contextual bandits in mobile health." In *Mobile Health*, pp. 495–517. Springer.

Wang CC, Kulkarni SR, Poor HV (2005). "Arbitrary side observations in bandit problems." *Advances in Applied Mathematics*, **34**(4), 903–938.

Whittle P (1979). "Discussion on:"Bandit processes and dynamic allocation indices"[JR Statist. Soc. B, 41, 148–177, 1979] by JC Gittins." *J. Roy. Statist. Soc. Ser. B*, **41**, 165.

Wilson RC, Geana A, White JM, Ludvig EA, Cohen JD (2014). "Humans use directed and random exploration to solve the explore–exploit dilemma." *Journal of Experimental Psychology: General*, **143**(6), 2074.

Affiliation:

Robin van Emden
Jheronimus Academy of Data Science
Den Bosch, the Netherlands
E-mail: robin@pwy.nl
URL: pavlov.tech

Maurits C. Kaptein
Tilburg University
Statistics and Research Methods
Tilburg, the Netherlands
E-mail: m.c.kaptein@uvt.nl
URL: www.mauritskaptein.com

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd