contextual: Simulating Contextual Multi-Armed Bandit Problems in R

Robin van Emden
JADS

Eric Postma
Tilburg University

Maurits Kaptein
Tilburg University

Abstract

Contextual bandit algorithms have been gaining in popularity due to their effectiveness and flexibility in the online evaluation of partial information 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 practical, standardized and extensible way. To help close this gap between analytical research and real-life application the current paper introduces the object-oriented R package **contextual**: a user-friendly and, through its clear object oriented structure, easily extensible framework that facilitates the parallel comparison of contextual and non-contextual Bandit policies by means of 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 but only learn about the best course of action by choosing one way or the other repeatedly, learning but one step at a time. In such situations, the basic premise stays the same for each renewed decision: do you stick to what you already know and receive an expected result ("exploit") or choose something you don't know all that much about and potentially learn something new ("explore")? As we all encounter such dilemma's on a daily basis, it is easy to come up with many examples - for instance:

- Do you feed your next coin to the one-armed bandit that paid out last time, or do you test your luck on another arm, on another machine?
- When going out to dinner, do you explore new restaurants, or do you exploit familiar ones?
- Do you stick to your current job, or explore and hunt around?
- Do I keep my current stocks, or change my portfolio and pick some new 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?

Every one of these issues represents another take on the same underlying dilemma: when to explore, versus when to exploit. To get a better grip on such decision problems, and to learn if and when specific strategies might be more successful than others, such explore/exploit dilemmas have been studied extensively under the umbrella of the "Multi-Armed Bandit" problem (MAB problem). Here, an algorithm or "policy" repeatedly selects one out of a limited set of options or "arms," each with its particular (hidden) payout distribution. Every time the policy selects another arm, it receives a reward from the "multi-armed bandit," which represents all available arms together with their hidden reward distributions. The policy itself continuously seeks to maximize its average rewards over time by balancing the exploration of arms with more uncertain payoffs with the exploitation of arms that offer the highest current expected payoff. Importantly, on each repeated choice, the policy only receives a reward for the chosen arm: he or she remains in the dark about the potential rewards of the unchosen arms.

A recent MAB generalization known as the *contextual* Multi-Armed Bandit (cMAB) builds on the previous formalization by adding one crucial element: contextual information. Such contextual multi-armed bandits are actually known by many different names in about as many different fields of research: as "bandit problems with side observations", "bandit problems with side information", "associative reinforcement learning", "reinforcement learning with immediate reward", "associative bandit problems", or "bandit problems with covariates". However, the term "contextual Multi-Armed Bandit," as coined by Langford and Zhang, seems both the most generally used and the most concise, so that is the term we will stick to in the current paper.

Still, however they are named, all cMAB policy differentiate themselves by definition from their MAB cousins in that can make use of features that reflect the current state of the world–features that can then be mapped onto available arms or actions. This access to side information makes cMAB algorithms even more relevant to many real-life decision problems than its MAB progenitors. 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 almost no choice exists without some contextual information that can be mined or mapped. So it may be no surprise that cMAB algorithms have found many applications: from recommender systems and advertising to health apps and personalized medicine–inspiring a multitude of new, often analytically derived bandit algorithms or policies, each with their strengths and weaknesses.

Regrettably, though cMAB algorithms have gained traction in both research and industry, 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 policies. There exist a few other frameworks that enable the analysis of offline datasets in some capacity, such as Microsoft’s Vowpal Wabbit, and the MAB focussed python package Striatum. But, as of yet, no extensible and widely applicable R package that can analyze and compare, respectively, K-armed, Continuum, Adversarial and Contextual Multi-Armed Bandit Algorithms on either simulated or 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 this 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. Which leads up to section 3, where we delve a little deeper into the implementation of the described classes.

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 $B = \{D_1, \ldots, D_k\}$, 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 $x_t = (x_{1,t}, \ldots, x_{d,t})$. Based on earlier payoffs, the **policy** then selects one of the **bandit** B's arms by choosing an action a $a_t \in \{1, \ldots, k\}$, 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 through some investigation of how these contexts, actions and rewards hang together. 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 $x_t = (x_{1,t}, \ldots, x_{d,t})$
2) Bandit B generates reward vector $r_t = (r_{t,1}, \ldots, r_{t,k})$
3) Policy π selects one of bandit B's arms $a_t \in \{1, \ldots, k\}$
4) Policy π gets reward r_{t,a_t} from bandit B and updates its arm-selection strategy with $(x_{t,a_t}, a_t, r_{t,a_t})$

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

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

This *cumulative reward* is directly related to *cumulative regret*, an oft used metric of policy performance. Cumulative regret is 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 }:

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

Though a policy's *cumulative reward* already offers some useful estimate of its learning performance, it is generally more informative to use a policy's *cumulative regret* as its performance measure. Firstly, cumulative regret offers normalization of a policy's performance. That is, with cumulative regret, it is possible to move a policy from one bandit to another who's rewards have been shifted by some arbitrary constant, and still arrive at the same total cumulative regret over T. Secondly, since good policies asymptotically approach that of a policy with the highest expected reward, their regret is expected to grow as a logarithm of T. In other words, 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.

See for example Figure 4 in section 4.3 for an illustrative example of how cumulative regret generally has nicer properties as a measure of a policy's performance when compared to cumulative reward. To be more specific, in this and other tables and figures, we mostly do not refer to the a policy's cumulative regret per se, but actually to their *expected* cumulative regret This is indicative of the fact that simulated rewards and actions are usually stochastic:

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

Where the expectation $\mathbb{E}[\cdot]$ is taken with respect to the random draw of both the rewards assigned by a bandit and the arms 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`[1]), 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`.
- `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 = 100)
history     <- simulator$run()
```

In these lines, we start out by instantiating the `Policy` subclass `EpsilonGreedyPolicy` as `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, our `bandit`'s probability of reward is set to respectively 0.9, 0.1 and 0.1 per arm. 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, the `Simulator` runs one hundred simulation, each with a different random seed, for one hundred time steps `t`.

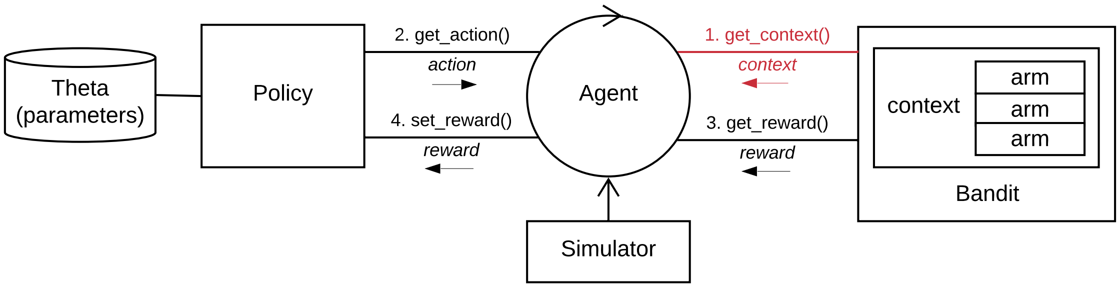

Figure 1: Diagram of contextual's basic structure. The context feature vector returned by get_context() (colored red in the figure) is only taken into account by cMAB policies, and is ignored by MAB policies.

On running the `Simulator`, it starts as many parallel worker processes as possible, each running another `agent` in parallel. Each of these `agents` then loops through four main function calls at each time step `t`. Though we delve deeper into the setup of each of the main contextual classes in section 3, the current overview allows 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)` that contains 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` decides 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`[1]). `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 = 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 the simulation, `Simulator` returns an `history` object that contains a complete log of all interactions, which can, among others, be printed, plotted, or summarized:

```
summary(history)

                cumulative_regret cumulative_reward
EpsilonGreedy   79.44             9.9
```

3. R6 class structure

Though section two's basic overview of contextual's class structure suffices for running predefined `Policies` and `Bandits` it lacks the detail necessary to extend contextual's classes. Since it is the contextual package'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 will give some more background information–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 would offer an easily extendible and open bandit framework together with an extensible bandit library presenting some of the best known and popular bandit algorithms. 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 in both academia and business offers precisely the kind of cross-pollination that **contextual** hopes to facilitate.

As R offers several different systems of object-orientation (R5, R6, S3, and S4) that meant we still needed to decide on a class system that would enable us to divide our package into clear, self-contained objects. In the end, on weighing the pros and cons of each class system, we decided to use the R6

[1] Here it is assumed that at each time step `t`, all information necessary to choose an arm is summarized using the limited set of parameters denoted θ_t, whose dimensionality is much smaller than of the log of all historical interactions.

system. First of all, because R6 uses reference semantics and is an encapsulated object-oriented system, where objects contain methods that can modify objects directly. These reference semantics make R6 classes instantly recognizable for developers with a background in Java or C++–in contrast to S3 and S4 classes, whose objects are not mutable, often making S3 and S4 classes more convoluted and verbose.

Additionally, 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, making **contextual** substantially less resource-hungry than they would otherwise have been–certainly not unimportant in a simulation package such as **contextual**.

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, as these are the classes that, in general, be the ones that will be extended most often. To further 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 6 on page 25 visualizes the structure of our package by showcasing the most important of **contextual**'s classes, attributes, and relationships at rest. The UML sequence diagram in figure Figure 6 on page 26, on the other hand, shows how **contextual**'s classes behave over time. This diagram depicts a basic overview of the sequence of function calls between **contextual**'s main objects in action.

Bandit

The abstract class `Bandit` is the superclass of any `Bandit` subclass that is to be implemented in **contextual**. As it is an abstract class, it declares methods but contains no implementation. That is, every `Bandit` class in the **contextual** package inherits from and has to implement the methods of by this class.

In practice, this implies that any `Bandit` subclass needs to set `self$k` to the number of arms, and `self$d` to the number of context features during its initialisation. On meeting this requirement, the `Bandit` is then required to implement `get_context()` and `do_action()`:

```
Bandit <- R6::R6Class(
  public = list(
    k              = NULL,  # number of arms (integer)
    d              = NULL,  # dimension of context feature (integer)
    precaching     = FALSE, # pregenerate context & reward matrices? (boolean)

    get_context = function(t) {
      stop("Bandit subclass needs to implement bandit$get_context()")
      # return a list with self$k, self$d and, where applicable, context vector X.
      list(k = n_arms, d = n_features, X = context)
    },
    get_reward = function(t, context, action) {
      stop("Bandit subclass needs to implement bandit$do_action()")
      # return a list with the reward and, if known, the reward of the best arm.
      list(reward = reward_for_choice_made, optimal = optimal_reward)
    },
```

```
    generate_bandit_data = function(n) {
      # called when precaching is TRUE. Pregenerates contexts and rewards.
      stop("Bandit subclass needs to implement bandit$generate_cache()
           when bandit$precaching is TRUE.",
    }
  )
)
```

Bandit's functions can be described as following:

- `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 repeated generation of these vectors. To facilitate this, during initialisation **contextual** calls `generate_bandit_data()` for every Bandit where `self$precaching` 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 k-armed bandit synthetically generates rewards based on a weight vector. It returns and empty context vector X.
- `ContextualBandit`: a basic contextual bandit that synthetically generates contextual rewards based on randomly set weights. It can simulate mixed user (cross-arm) and article (arm) feature vectors, following its parameters k, d and num_users.
- `ContinuumBandit`: a basic example of a continuum bandit.
- `SyntheticBandit`: an example of a more complex and versatile synthetic bandit, that pregenerates its context and reward vectores.
- `LiBandit`: a basic example of a bandit that makes use of offline data - here, an implementation of Li(2232)

Each of these bandits can be used to test policies without further ado. However, they can also serve as superclasses for new custom `Bandit` subclasses. Or as templates for new `Bandit` implementation(s) that directly subclass the `Bandit` superclass.

Policy

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

```
Policy <- R6::R6Class(
  public = list(
    name          = "",
    action        = NULL,     # action list
    theta         = NULL,     # list of all parameters theta
    theta_to_arms = NULL,     # theta to arms list
    initialize = function(name = "Not implemented") {
      self$name   <- name     # each policy has a name
      self$theta  <- list()   # list that keeps track of all parameter values
      self$action <- list()   # initiatlisation of action list for internal use
    },
    get_action = function(t, context) {
      # chooses arm based on theta and context, returns its index in action$choice
      stop("Policy$get_action() has not been implemented.", call. = FALSE)
    },
    set_reward = function(t, context, action, reward) {
      # updates parameters in theta based on reward awarded by bandit to chosen arm
      stop("Policy$set_reward() has not been implemented.", call. = FALSE)
    },
    set_parameters = function() {
      # policy parameters (not theta!) initialization happens here
      stop("Policy$set_parameters() has not been implemented.", call. = FALSE)
    },
    initialize_theta = function() {
      # implementation not shown - 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` keeps track of the state, and 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` and `theta` 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 their correct order:

```
Agent <- R6::R6Class(
  public = list(
    #...
    do_step = function() {
      private$t <- private$t + 1
      context = bandit$get_context(private$t)
      action  = policy$get_action (private$t, context)
      reward  = bandit$get_reward (private$t, context, action)
      theta   = policy$set_reward (private$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:

```r
history <- Simulator$new(agents = agent, horizon = 100, simulations = 100)$run()
```

To specify how to run a simulation and how 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 the context vectors `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 is possible to plot() a History object, summarize() it, or, among others, obtain either a data.frame() or a data.table() from any History instance:

```r
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.
- `grid`: plots a combination of the previous plots in a 2x2 grid.
- `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 plot(). In either case, make sure to specify (at least) a History subclass and one of the plot types specified above:

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

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

Some other plot arguments:

4. Implementing and extending Policy and Bandit subclasses

Though section 3 provides an introduction to all of contextual's main classes, in practice, most developers and researchers will mostly focus on subclassing policies and bandits of their own. Therefore, the current section first demonstrates how to implement some well-known bandit algorithms, and, secondly, how to create Policy and Bandit sub-subclasses.

4.1. Epsilon First

In this non-contextual algorithm, also known as AB(C) testing, a pure exploration phase is followed by a pure exploitation phase. The Epsilon First policy is equivalent to the setup of a randomized controlled trial (RCT): a study design where people are allocated at random to receive one of several clinical interventions. One of these interventions is the control. This control can be standard practice, a placebo, or no intervention at all. On completion of the RCT, the best solution at that point is then suggested to be the superior "evidence-based" option for everyone, at all times.

To get an idea of how straightforward it can be to go from a high-level policy description in pseudocode, the format that is most generally used to formally describe and share bandit policies in the literature, to an R contextual Policy subclass, first a pseudocode description of the Epsilon First algorithm:

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

Next, its conversion from the above pseudocode to an EpsilonFirstPolicy class:

```
EpsilonFirstPolicy <- R6::R6Class(
  public = list(
    first = NULL,
    initialize = function(first = 100, name = "EpsilonFirst") {
      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
    }
  )
)
```

At that point, it is easy to run any policy by assigning it to an agent together with one of contextual's Bandit subclasses. Then assign and run the agent via a Simulator instance:

```
library("contextual")

horizon            <- 100
simulations        <- 100
weights            <- c(0.9, 0.1, 0.1)

policy             <- EpsilonFirstPolicy$new(first = 50, name = "EFirst")
bandit             <- SyntheticBandit$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(5, 5, 1, 1))
plot(history, type = "cumulative", grid = TRUE)
plot(history, type = "arms", grid = TRUE)
```

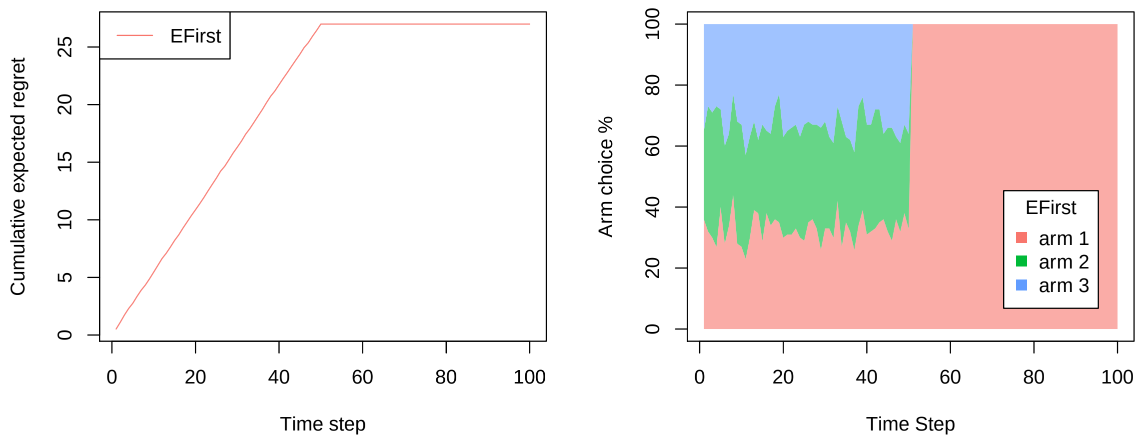

Figure 2: Epsilon First

4.2. The Epsilon Greedy Policy

Contrary to the previously introduced Epsilon First policy, Epsilon Greedy algorithms do not divide exploitation and exploration into two strictly separate phases. On the contrary: an Epsilon Greedy policy starts exploring and exploiting from step one, and never stops balancing the two strategies. That is, an Epsilon Greedy policy generally mostly plays the currently best-known arm. At those times, the policy acts "greedy," as it exploits the best available option up to that point in time. However, a fraction epsilon ϵ of the time, the policy picks some arm at random, exploring if another arm might not be better after all.

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, name = "EGreedy") {
      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
    }
  )
)
```

Then, just run it again:

```
library("contextual")

horizon            <- 100
simulations        <- 100
weights            <- c(0.9, 0.1, 0.1)

policy             <- EpsilonGreedyPolicy$new(epsilon = 0.1, name = "EG")
bandit             <- SyntheticBandit$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(5, 5, 1, 1))
plot(history, type = "cumulative", grid = TRUE)
plot(history, type = "arms", grid = TRUE)
```

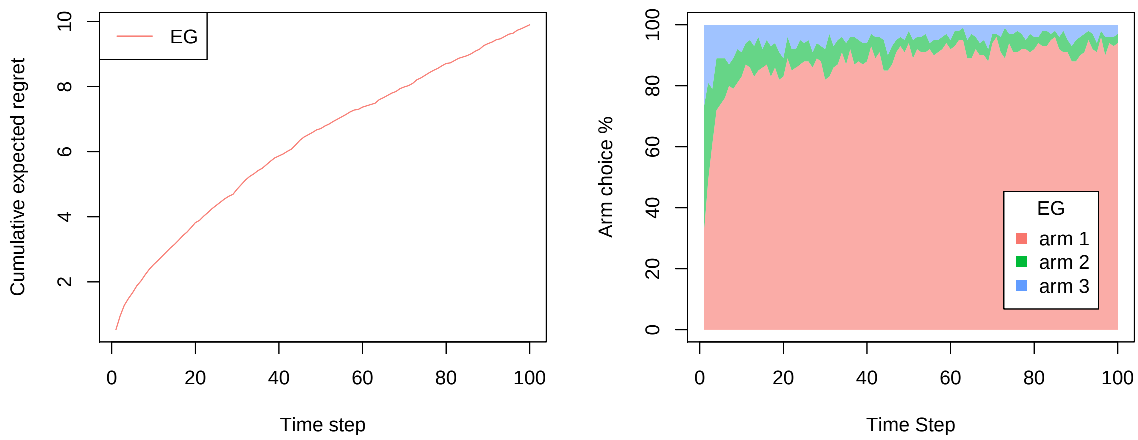

Figure 3: Epsilon Greedy

4.3. 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 on a site visitor's country of residence may influence which out of several articles (or arms) will be most popular when placed on the website's front page.

Here, we show how to implement and evaluate probably one of the most cited out of all contextual policies, known as 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 is not all that hard to implement as yet another Bandit subclass.

The algorithm first runs 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.

Now, let's first take a look at a high-level description the LinUCB Disjoint algorithm 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, translate the above pseudocode into a well organised Bandit subclass again:

```
#' @export
LinUCBDisjointPolicy <- R6::R6Class(
  public = list(
    alpha = NULL,
    initialize = function(alpha = 1.0, name = "LinUCBDisjoint") {
      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
    }
  )
)
```

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

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

weights      <- matrix(c(0.9, 0.1, 0.1,      # d=1   -> rows represent
                        0.1, 0.9, 0.1,      # d=2      context features
                        0.1, 0.1, 0.9),     # d=3

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

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

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

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

par(mfrow = c(1, 2),mar = c(5, 5, 1, 1))
plot(history, type = "cumulative", regret = FALSE, grid = TRUE)
plot(history, type = "cumulative", grid = TRUE)
```

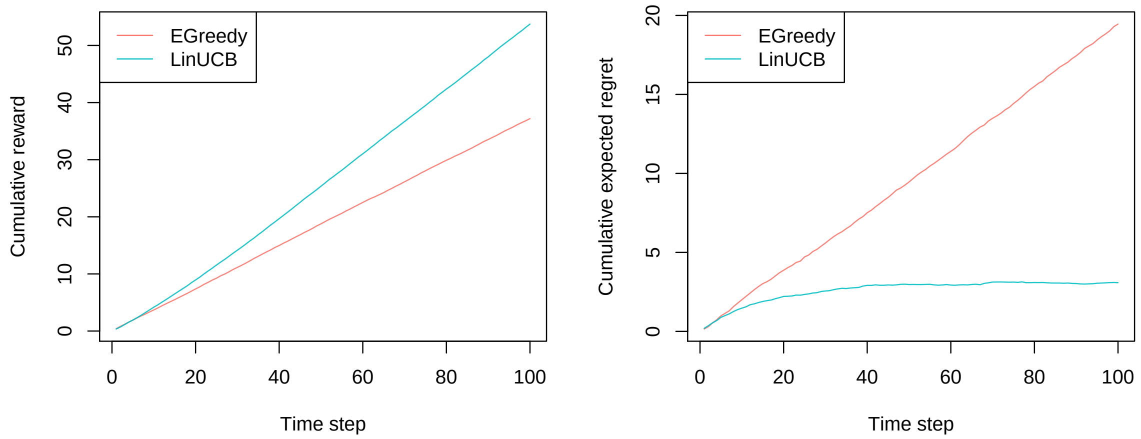

Figure 4: The contextual LinUCB algorithm with linear disjoint models, following Li et al. (2010), compared to a non-contextual basic EpsilonGreedy policy tested on data generated by a contextual synthetic Bandit for two performance measures: left cumulative reward, right expected cumulative regret.

4.4. Extending non-Policy classes

Of course, the extensibility of contextual does not limit itself to the subclassing of Policies. Through its R6 based object system, it is also easy to extend any other **contextual** super- or subclass. Below, we demonstrate how to make use of that extensibility through the implementation of a PoissonRewardBandit extending **contextual**'s BasicBandit class, and of a PoissonRewardBandit version of the Epsilon Greedy policy that we introduced in section 4.2.

```
PoissonRewardBandit <- R6::R6Class(
  "PoissonRewardBandit",
  # Class extends BasicBandit
  inherit = BasicBandit,
  public = list(
    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(
  "EpsilonGreedyAnnealingPolicy",
  # Class extends EpsilonGreedyPolicy
  inherit = EpsilonGreedyPolicy,
  portable = FALSE,
  public = list(
    # 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$new(0.1, "EG Annealing"), bandit),
                    Agent$new(EpsilonGreedyAnnealingPolicy$new(0.1, "EG"), bandit))
simulation  <- Simulator$new(agents, horizon, simulations, do_parallel = FALSE)

history     <- simulation$run()

par(mfrow = c(1, 2),mar = c(5, 5, 1, 1))
plot(history, type = "cumulative", grid = TRUE)
plot(history, type = "average", regret = FALSE, grid = TRUE)
```

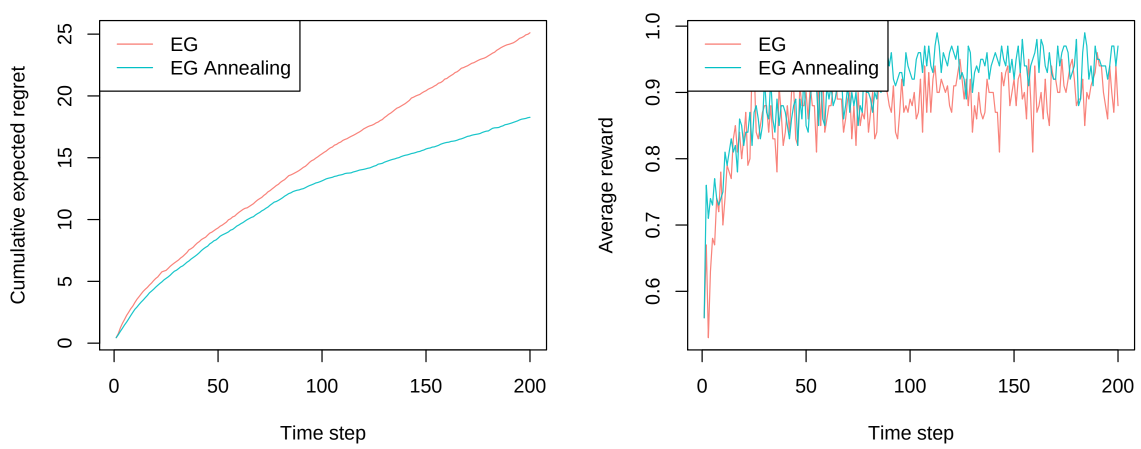

Figure 5: Extending BasicBandit and EpsilonGreedyPolicy

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:

```
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)

data_dir        <- "data/"
website_data    <- data.table::fread(paste0(data_dir,"persuasion.csv"))
horizon         <- 350000L
simulations     <- 100L
bandit          <- BasicLiBandit$new(website_data, arms = 4) # four types of webpages can be presen
agent           <- Agent$new(UCB1Policy$new(), bandit)       #
history         <- Simulator$new(agent, horizon, simulations, reindex_t = TRUE)$run()

plot(history, type = "cumulative", regret = FALSE, rate = TRUE, ylim = c(0.0105,0.015))
plot(history, type = "arms")
```

Offline evaluation through DoublyRobustBandit

*** insert algorithm ***

*** insert code ***

6. Replication of existing studies

Here we replicate some papers with a huge offline dataset..

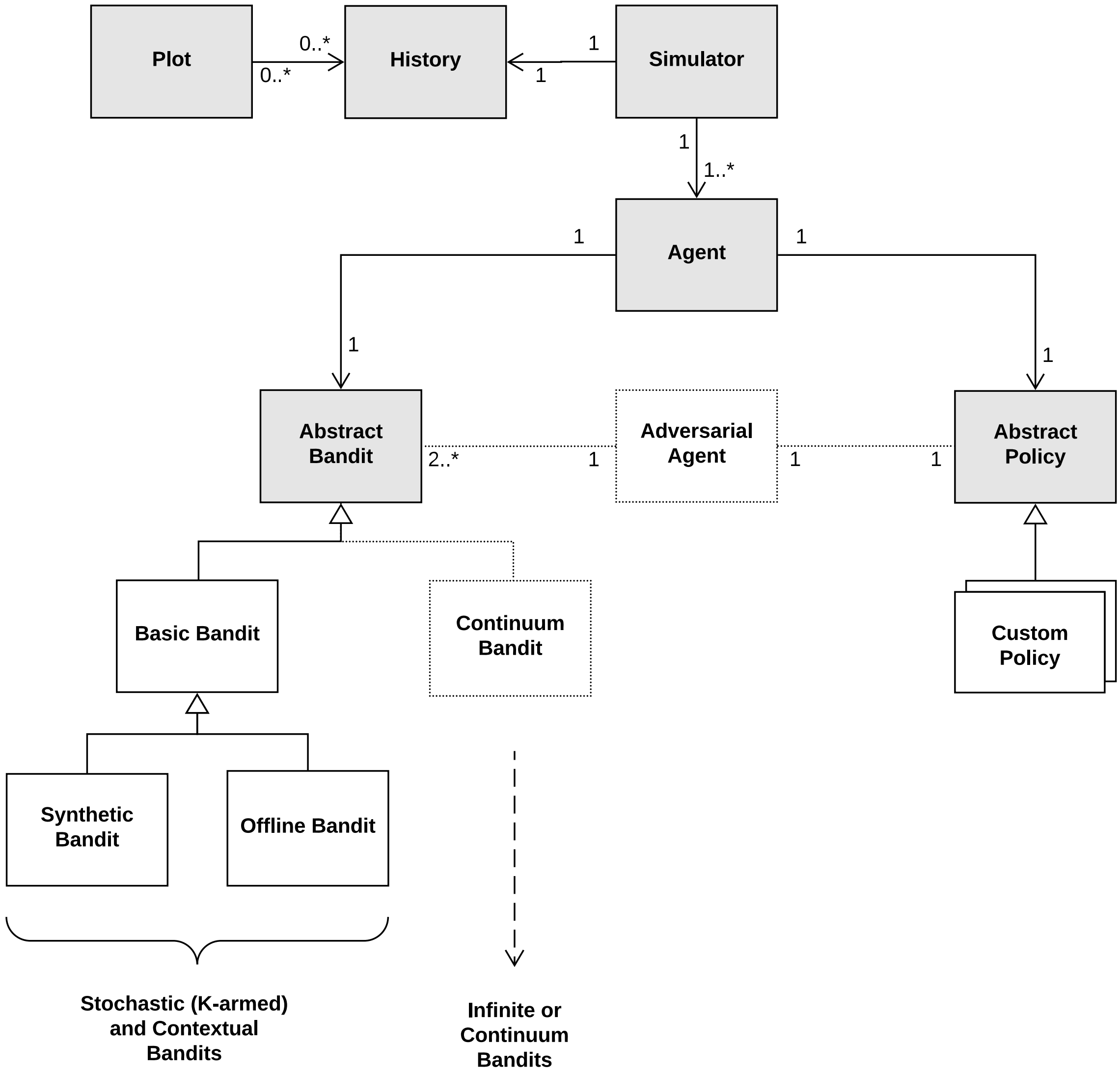

Figure 6: **contextual** UML Class Diagram

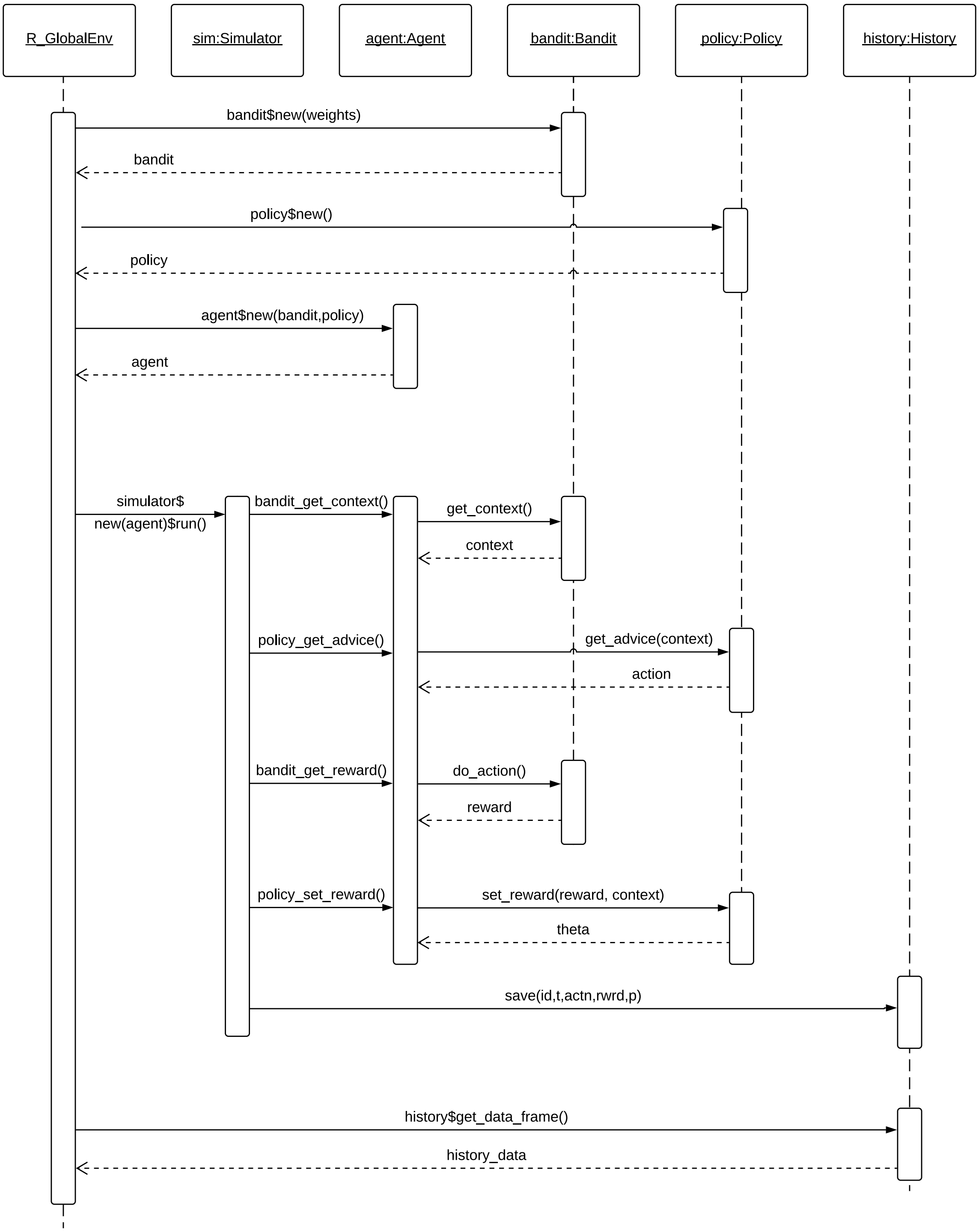

Figure 7: **contextual** UML Sequence Diagram

7. Special features

For instance, quantifying variance..

8. The art of optimal parallelisation

There is a very intersting trade of between the amount of parallelisation (how many cores, nodes used) the resources needed to compute a certain model, and the amount of data going to and fro the cores.

PERFORMANCE DATA ———————————————————————

on 58 cores: k3*d3 * 5 policies * 300 * 10000 –> 132 seconds

on 120 cores: k3*d3 * 5 policies * 300 * 10000 –> 390 seconds

—

on 58 cores: k3*d3 * 5 policies * 3000 * 10000 –> 930 seconds

on 120 cores: k3*d3 * 5 policies * 3000 * 10000 –> 691 seconds

9. Extra greedy UCB

Ladila bladibla.

10. Conclusions

Placeholder... the goal of a data analysis is not only to answer a research question based on data but also to collect findings that support that answer. These findings usually take the form of a table, plot or regression/classification model and are usually presented in articles or reports.

11. Acknowledgments

Thanks go to CCC.

Affiliation:

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

Eric O. Postma
Tilburg University
Communication and Information Sciences
Tilburg, the Netherlands
E-mail: e.o.postma@tilburguniversity.edu

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