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 website editor, do you keep the same article at the top of your news site, or alternate between articles?

- As a doctor, do you treat your patients with tried and tested interventions, or do you prescribe new and promising experimental treatments?

Let's delve a little deeper into one last example. You received some gambling money, and now have to decide which out of k slot machines or "one-armed bandits"[1] to play[2]. In this scenario, each "bandit" is similar, though differing 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 nothing more than 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 (Bubeck, Cesa-Bianchi *et al.* 2012).

A recent MAB generalization known as the *contextual* Multi-Armed Bandit (cMAB) extends it by adding one crucial element: contextual information (Langford and Zhang 2008). 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 2006b), 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 choose the same type of restaurants in summer and winter? Do you prescribe the same treatments to male and female patients? Do you place the same news story at the top featured position of your website for both young and old visitors? Probably not—in the real world, it appears no choice exists without at least some contextual information to be mined or mapped. So it may be no surprise that cMAB algorithms have found applications 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.

---

[1]The term "one-armed bandit" 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.

Regrettably, though cMAB algorithms have gained both academic and commercial acclaim, comparisons on simulated, and, importantly, real-life, large-scale offline datasets (Li, Chu, Langford, and Wang 2011) 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 some other frameworks that enable the analysis and comparison of either on- or offline datasets, such as Microsoft's **Vowpal Wabbit** (Langford, Li, and Strehl 2007), the online evaluation platform StreamingBandit (Kaptein and Kruijswijk 2016), and the MAB focused Python packages **Striatum** (NTUCSIE-CLLab 2018) and **SMPyBandits** (Besson 2018). But, as of yet, no extensible and widely applicable R (R Core Team 2018) package that can analyze and compare, respectively, basic multi-armed, Continuum (Agrawal 1995) and Contextual Multi-Armed Bandit Algorithms on both simulated and offline data.

In the following paper, we further introduce the **contextual** package. Section 2 presents a formal definition of the contextual Multi-Armed Bandit problem and shows how this formalization can be translated into a clear and concise object-oriented implementation. Section 3 gives an overview of **contextual**'s predefined contextual and non-contextual bandit and policy classes and shows how to apply these to solve some basic (c)MAB problems. Section 4 delves a little deeper into the implementation of pkgcontextual's core classes—in preparation for Section 5, which shows how to extend these classes to be able to create custom bandits and policies. Section 6 focusses on the analysis of offline "partial label" datasets. Section 7 brings all of the previous sections together in the example of a partial replication of a seminal contextual bandit paper. We conclude with some comments on the current state of the package and potential future enhancements.

This organization offers the reader several ways to peruse the current paper. If you have a passing knowledge of R and want to run simulations based on **contextual**'s default bandits and policies, the current introduction plus section 3 should be able to get you up and running. If you are looking for a more formal introduction, read section 2 as well. Include section 6 and possibly 7 if you also want to run one of **contextual**'s policies on an offline dataset. Finally, if you know your way around R and would like to extend **contextual** to run custom bandits and policies, it is probably best to read the whole paper—with a focus on sections 4 and 5.

2. Formalization and implementation

In the current section, we first introduce a more formal definition of the contextual Multi-Armed Bandit problem. Next, we translate this formalization into a clear and concise implementation.

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). 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 **bandit** B's arms by choosing an action $a_t \in \{1, \ldots, k\}$, and receives reward $r_{a_t,t}$, the expectation of which depends both on 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

parameters summarizes all historical interactions $H_{t'} = (x_{t,a_t}, a_t, r_{t,a_t})$ over $t = \{1, \ldots, t'\}$, ensuring that the dimensionality of $\theta_{t'} \ll H_{t'}$.

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

Schematically, for each round $t = \{1, \ldots, T\}$:

1) Policy π observes state of the world as contextual feature vector $x_t = (x_{1,t}, \ldots, x_{d,t})$

2) Policy π selects one of bandit B's arms $a_t \in \{1, \ldots, k\}$

3) Bandit B generates reward vector $r_t = (r_{t,1}, \ldots, r_{t,k})$

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 $t = \{1, \ldots, T\}$

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

The most popular performance measure for bandit policies ([Kuleshov and Precup 2014]) is *cumulative expected 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 chosen actions a^π for every t over $t = \{1, \ldots, T\}$:

$$\mathbb{E}\left[R_T^\pi\right] = \mathbb{E}\left[\max_{a=1,\ldots,k} \sum_{t=1}^{T}(r_{a,x_t}) - \sum_{t=1}^{T}(r_{a_t^\pi, x_t})\right] \tag{2}$$

Where expectation $\mathbb{E}[\cdot]$ is taken with respect to random draws of both rewards assigned by a bandit and arms as selected by a policy ([Zheng and Hua 2016]).

2.2. Implementation

We set out to develop an implementation that stays close to the previous formalization but also offers an easily accessible programming interface. 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 1):

- `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 `Agent`s 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 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 a 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`.
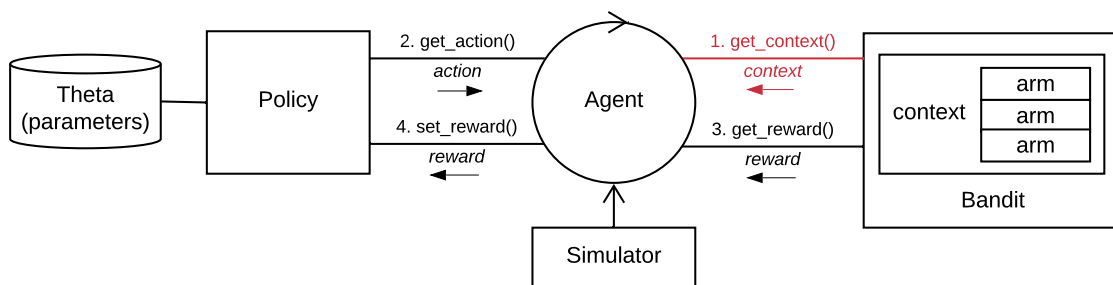

Figure 1: Diagram of **contextual**'s basic structure. The context feature vector returned by get_context() (colored red in the figure) is only taken into account by cMAB policies, and is ignored by MAB policies.

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 parallel 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 a `history` object that contains a complete log of all interactions, which can then, for example, be 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. Basic Usage

The current section offers an overview of **contextual**'s pre-implemented bandits and policies and explains how to run a basic simulation.

3.1. Implemented Policies and Bandits

Though **contextual** comes into its own when you start building your own bandit and policy classes, you can also run simulations with its built-in bandits and policies. See Table 1 for an overview of the currently implemented policies and Table 2 for an overview of all currently implemented bandits.

	ϵ-**Greedy**	**UCB**	**Thomspon Sampling**	**Other**	**Special**
MAB	ϵ-Greedy[1] ϵ-First	UCB1[2] UCB-tuned[3]	Thompson Sampling[4] BootstrapTS[5]	Softmax[6] Gittins[7]	Random Oracle
cMAB	Epoch-Greedy[9]	LinUCB[10] COFIBA[11]	LinTS[12] LogitBTS[13]		LiF[8]

[1] Sutton and Barto (1998) [2] Auer *et al.* (2002) [3] Auer *et al.* (2002) [4] Agrawal and Goyal (2011) [5] Eckles and Kaptein (2014) [6] Vermorel and Mohri (2005) [7] Brezzi and Lai (2002) [8] Kaptein *et al.* (2016) [9] Langford and Zhang (2008) [10] Li *et al.* (2010) [11] Li (2016) [12] Agrawal and Goyal (2012) [13] Eckles and Kaptein (2014)

Table 1: An overview of **contextual**'s predefined policy classes.

MAB	**cMAB**	**Offline**	**Continuous**
BasicBandit	BasicContextualBandit MixedContextualBandit	LiSamplingOfflineBandit DoublyRobustOfflineBandit	ContinuumBandit

Table 2: An overview of **contextual**'s predefined bandit classes.

3.2. Running a Simulation

To demonstrate how to run a basic simulation with **contextual**'s pre-implemented bandits and policies, we will compare two of contextual's pre-implemented polices in a scenario where each policy seeks to optimize the click-through rate of two different online ads over the course of 300 impressions.

The first policy is known as an Epsilon First policy or an A/B test. This policy will explore both ads at random for a preset period and from thereon only exploit the ad with the best click-through rate.

The other policy, known as an Epsilon Greedy policy (Sutton and Barto 1998), will explore one of the ads at random twenty percent of the time and exploits the ad with the best current click-through rate eighty percent of the time.

What policy is most effective in this basic scenario?

We can answer this question by running the following simulation:

```
# Load and attach the contextual package.
library(contextual)
# Define for how long the simulation will run.
horizon              <- 300
# Define how many times to repeat the simulation.
simulations          <- 400
# Define the probability that each ad will be clicked.
click_probabilities <- c(0.8, 0.2)
# Initialize a SyntheticBandit, which takes probabilites per arm for an argument.
bandit               <- SyntheticBandit$new(weights = click_probabilities)
# Initialize an EpsilonGreedyPolicy with a 20% exploiration rate.
eg_policy            <- EpsilonGreedyPolicy$new(epsilon = 0.2)
# Initialize an EpsilonFirstPolicy with a 50 step exploration period.
ef_policy            <- EpsilonFirstPolicy$new(first = 50)
# Initialize two Agents, binding each policy to a bandit.
ef_agent             <- Agent$new(ef_policy, bandit)
eg_agent             <- Agent$new(eg_policy, bandit)
# Assign both agents to a list.
agents               <- list(ef_agent, eg_agent)
# Initialize a Simulator with the agent list, horizon, and number of simulations.
simulator            <- Simulator$new(agents, horizon, simulations)
# Now run the simulator.
history              <- simulator$run()
# And plot the cumulative reward rate (equals click-through rate)
plot(history, type = "cumulative", regret = FALSE, rate = TRUE)
```

As can be seen in the left plot in Figure 3.2, the Epsilon First policy is the clear winner in this particular scenario. We can, however, make some minor changes to the scenario and end up with a very different outcome. In the script below we set one ad to be more effective in one age group, and the other to be more effective in a second age group. Next, we add a contextual policy (LinUCB, Li *et al.* (2010)), and run the simulation:

```
#                                  +-----+----------> two different ads
#                                  |     |
click_probabilities <- matrix(  c(  0.8,  0.2,    # --> context: older (p=.5)
                                    0.2,  0.8  ), # --> context: young (p=.5)

                   nrow = 2, ncol = 2, byrow = TRUE)

# Initialize a SyntheticBandit with the *contextual* click probabilities
context_bandit     <- SyntheticBandit$new(weights = click_probabilities)
# Initialize a contextual LinUCBDisjointPolicy with an alpha of 0.6
lu_policy          <- LinUCBDisjointPolicy$new(0.6)
# Initialize three Agents, binding each policy to a bandit.
ef_agent           <- Agent$new(ef_policy, context_bandit)
eg_agent           <- Agent$new(eg_policy, context_bandit)
lu_agent           <- Agent$new(lu_policy, context_bandit)
# Assign all agents to a list.
agents             <- list(ef_agent, eg_agent, lu_agent)
# Horizon can be much shorter here
horizon            <- 40
```

```r
# Initialize a Simulator with the agent list, horizon, and number of simulations.
simulator          <- Simulator$new(agents, horizon, simulations)
# Now run the simulator.
history            <- simulator$run()
# And plot the cumulative reward rate again.
plot(history, type = "cumulative", regret = FALSE, rate = TRUE)
```
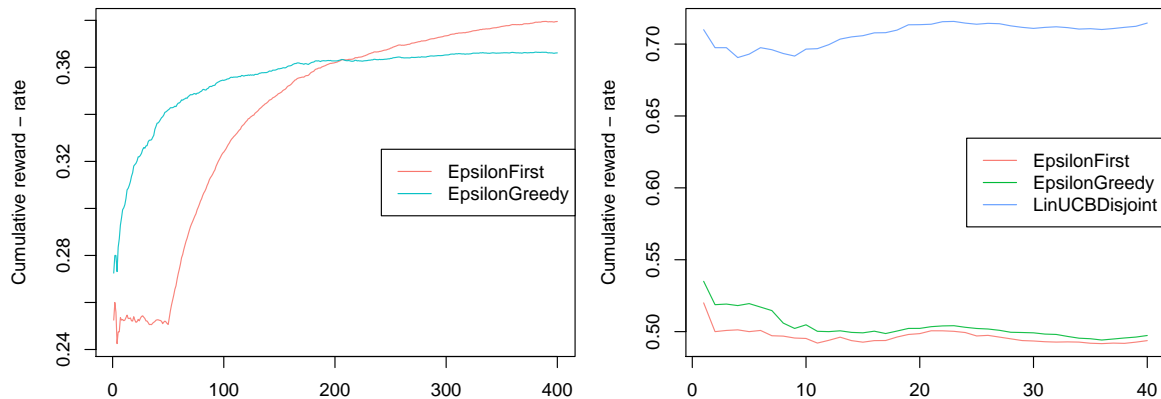

Figure 2: To the left, the cumulative reward rate (or click-through rate) of a non-contextual bandit scenario comparing Epsilon First and Epsilon Greedy policies—to the right the cumulative reward rate of a contextual bandit scenario comparing LinUCB, Epsilon First and Epsilon Greedy policies.

As can be seen in the right plot in Figure 3.2, this time around, neither non-contextual policy (Epsilon Greedy nor Epsilon First) does better than chance, as on average, both arms are equally likely to produce a click. On the other hand, the "LinUCB" policy does very well, as this contextual policy is able to map its rewards to the observed contextual features. Of course, neither of the simulations in the current section is very realistic. One way to generate more realistic Bandit data would be to write a Bandit subclass with a more complex generative model. How to create such a bandit subclass is delineated in section 5. Another option would, for example, be to evaluate policies on an offline dataset—more on that in section 6.

4. R6 class structure

Since it is **contextual**'s explicit goal to offer researchers and developers an easily extensible framework to develop, evaluate 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 Chang (2017) and on each of the six previously introduced core **contextual** classes.

4.1. R and the R6 Class System

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 [5] 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.

4.2. 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 39 visualizes **contextual**'s static object model, showing how its classes inherit from, and interface with, each other. The UML sequence diagram in figure Figure 11 on page 40, 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)
```

---

[5]In object-oriented programming, the developer compartmentalizes data into objects, whose behavior and contents are described through the declaration of classes. Its benefits include reusability, refactoring, extensibility, ease of maintenance and efficiency. See Wickham (2014) for more information of the use of OOP in R.

```
    # 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.

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` subclass 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`—though this can be changed by setting either `save_context` and `save_theta` arguments set to TRUE.

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 offline Bandits. If `continuous_counter` is set to TRUE, the current `Simulator` iterates over all rows in a dataset 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, or obtain, for example, 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 ways to plot it, each optimized to be able to plot gigabytes worth of data, quickly:

- `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.

- `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:

```r
# 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)
```

Multiple agents can be combined into one plot, and multiple plots can be combined by turning off the default par formatting of the Plot class and redefining it through, for example par(mfrow, mar) and such. A script that generates some example plots that make use of these features:

```r
bandit              <- SyntheticBandit$new(weights = c(0.9, 0.1, 0.1))
agents              <- list(Agent$new(RandomPolicy$new(), bandit),
                            Agent$new(OraclePolicy$new(), bandit),
                            Agent$new(ThompsonSamplingPolicy$new(1.0, 1.0), bandit),
                            Agent$new(Exp3Policy$new(0.1), bandit),
                            Agent$new(GittinsBrezziLaiPolicy$new(), bandit),
                            Agent$new(UCB1Policy$new(), bandit))
history             <- Simulator$new(agents, horizon = 100, simulations = 1000)$run()
par(mfrow = c(3, 2), mar = c(5, 5, 1, 1))
plot(history, type = "cumulative", use_colors = FALSE, no_par = TRUE)

plot(history, type = "cumulative", regret = FALSE, legend = FALSE,
     limit_agents = c("UCB1"), traces = TRUE, no_par = TRUE)

plot(history, type = "cumulative", regret = FALSE, rate = TRUE, ci = "sd",
     limit_agents = c("Exp3", "ThompsonSampling"),
     legend_position = "bottomright", no_par = TRUE)

plot(history, type = "cumulative", rate = TRUE, plot_only_ci = TRUE,
     ci = "var", smooth = TRUE, limit_agents = c("UCB1", "GittinsBrezziLai"),
     legend_position = "topright", no_par = TRUE)

plot(history, type = "average", ci = "ci", regret = FALSE, interval = 10,
     smooth = TRUE, legend_position = "bottomright", no_par = TRUE)

plot(history, limit_agents = c("ThompsonSampling"), type = "arms",
     interval = 20, no_par = TRUE)
par(mfrow = c(1, 1))
```
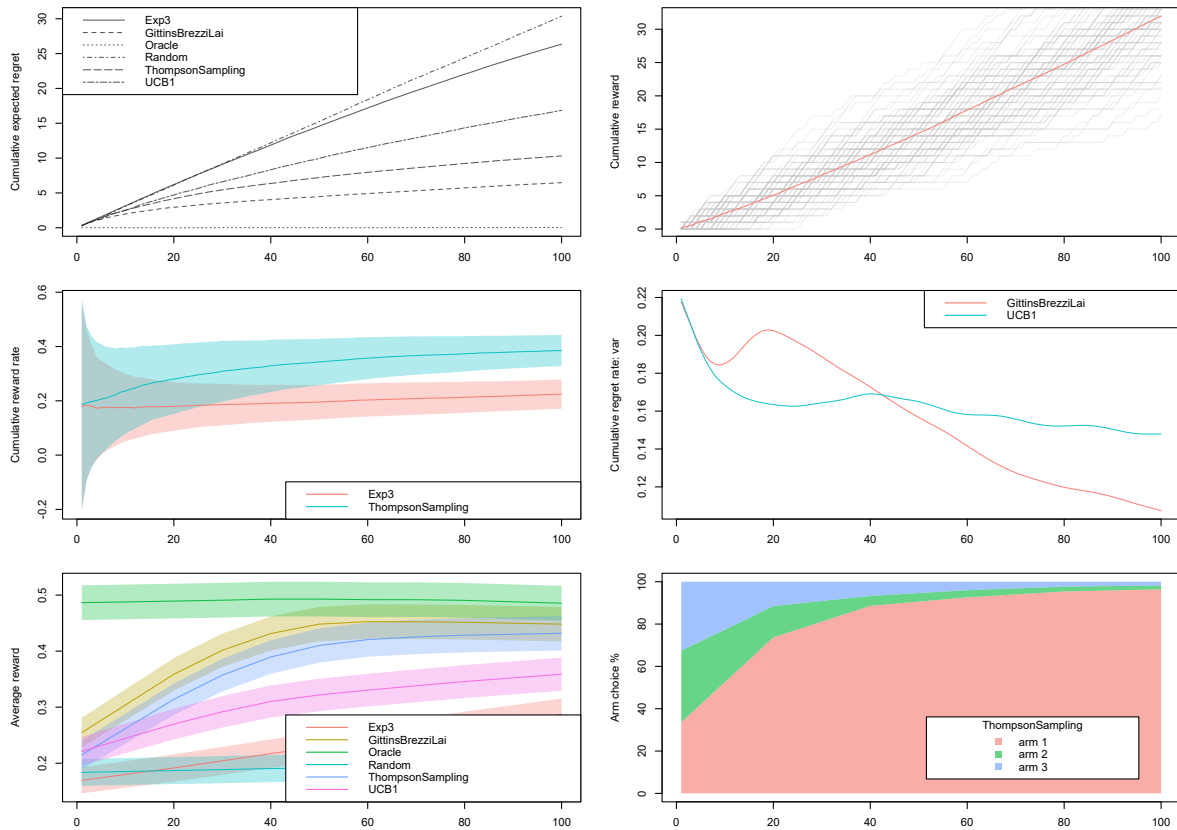
5. Implementing and extending Policy and Bandit subclasses

Though section 3 provides a first 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.

5.1. 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 $1 - p$. 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 p_a, the expected reward of the arm.

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

```r
BasicBandit <- R6::R6Class(
  inherit = Bandit,
  public = list(
    initialize  = function(weights = NULL) {
      self$set_weights(weights)                    # arm weight vector
```

```r
      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.

5.2. 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 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:

```r
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:

```r
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)
```


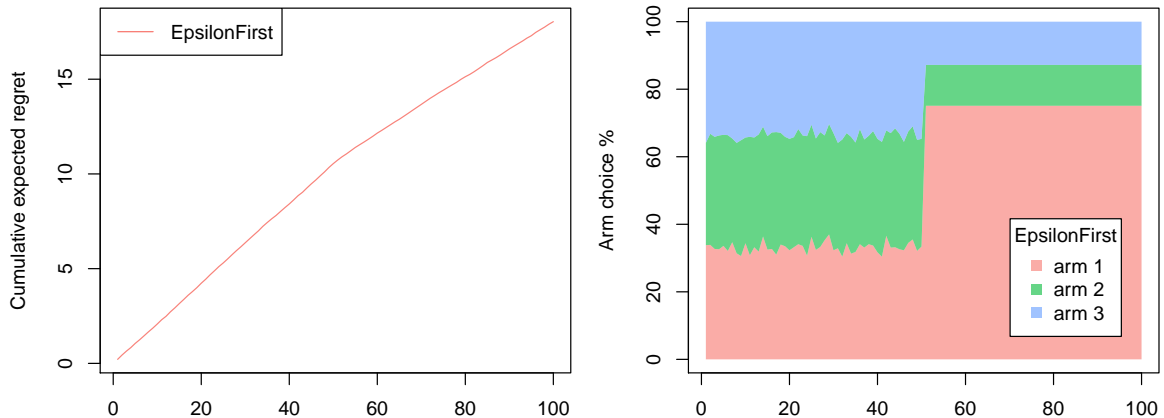Figure 3: Epsilon Greedy

5.3. The Epsilon Greedy Policy

Contrary to the previously introduced Epsilon First policy, an Epsilon Greedy algorithm Sutton and Barto (1998) 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 $unif(0, 1) > \epsilon$ **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)
```
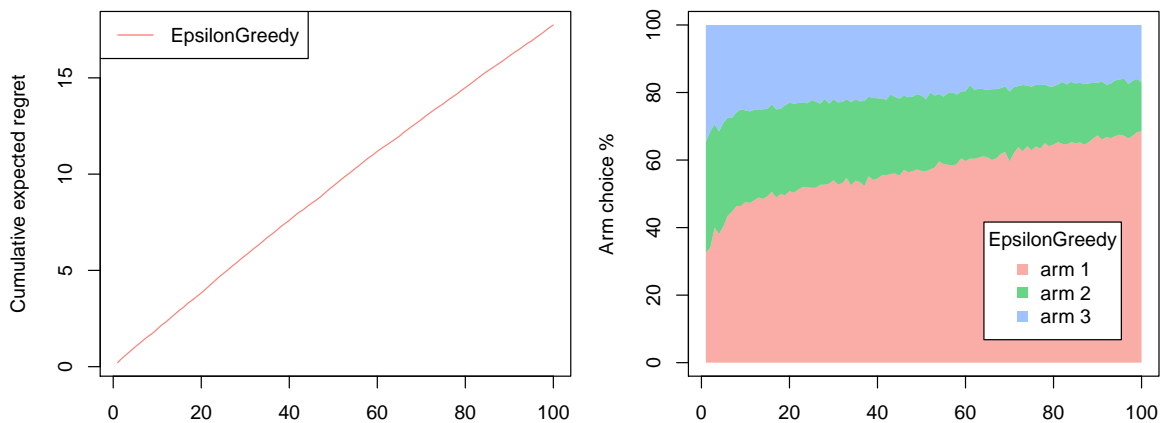

Figure 4: Epsilon Greedy

5.4. 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 make use of side inform-

ation 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 Li *et al.* (2010). 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, following Algorithm 1 from Li *et al.* (2010):

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:

```r
#' @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)
```
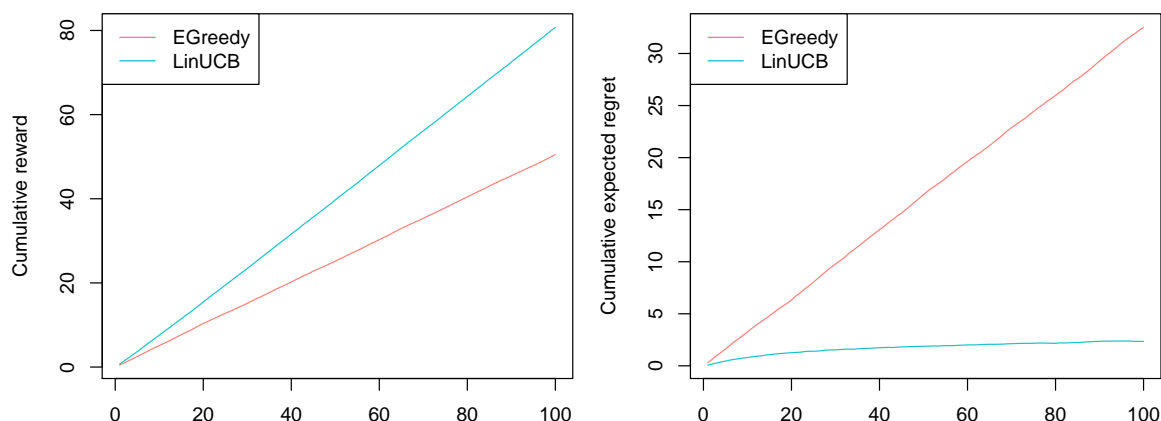

Figure 5: The contextual LinUCB algorithm with linear disjoint models, following Li et al. (2010), compared to a non-contextual basic EpsilonGreedy policy tested on data generated by a contextual synthetic Bandit for two performance measures: left cumulative reward, right expected cumulative regret.

5.5. 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.

```
PoissonRewardBandit <- R6::R6Class(
  # Class extends BasicBandit
  inherit = BasicBandit,
  public = list(
    class_name = "PoissonRewardBandit",
```

```r
    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)
```
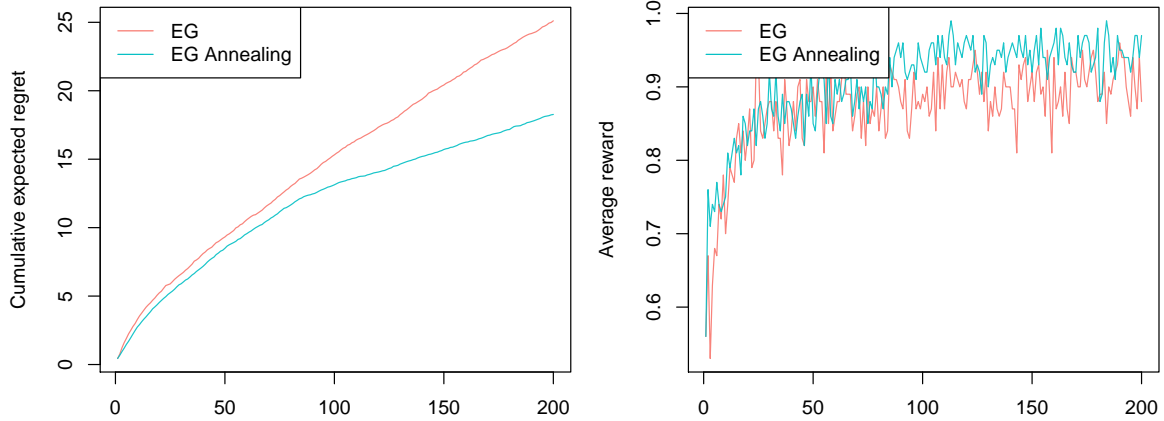
Figure 6: Extending BasicBandit and EpsilonGreedyPolicy

6. Offline evaluation

Though it is, as demonstrated in the previous section, relatively easy to create basic synthetic Bandits to evaluate 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 (Strehl, Li, Wiewiora, Langford, and Littman 2006a). 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 (Li, Chu, Langford, Moon, and Wang 2012; Li *et al.* 2011).

It would, of course, be possible to evaluate policies by running them in a live setting. Such live evaluations would 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 evaluate multiple policies at the same time with for same user (Mandel, Liu, Brunskill, and Popovic 2016). Using live data is usually also much slower than an offline evaluation, as online evaluations are dependent on active user interventions (Tewari and Murphy 2017). 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 (Bastani and Bayati 2015).

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 datasets 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 datasets for supervised learning, and more (Li *et al.* 2011).

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 dataset. In other words, offline data is "partially labeled" in respect to evaluated Bandit policies (Strehl, Langford, Li, and Kakade 2010). 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 datasets.

6.1. 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 (Li *et al.* 2012, 2011). In pseudocode, following Algorithm 2 from (Li *et al.* 2011):

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**

      $h_t \leftarrow$ 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

Below, an example of Algorithm 4 converted to pkgcontextual BasicBandit subclass BasicLiBandit run on an offline simulation. The offline dataset used in this example contains the click-through data of an online web store. Each of its 350000 rows represents one page view, with click (1) or non-click (0) information plus contextual information on which out of four "persuasive strategies" was used to persuade visitors to click through on that particular page. See Figure 8 for the resulting click-through rate over time.

```r
BasicLiBandit <- R6::R6Class(
  "BasicLiBandit",
  inherit = BasicBandit,
  portable = TRUE,
  class = FALSE,
  private = list(
    S = NULL
  ),
  public = list(
    initialize   = function(data_stream, arms) {
      self$k <- arms
```

```r
      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))
```
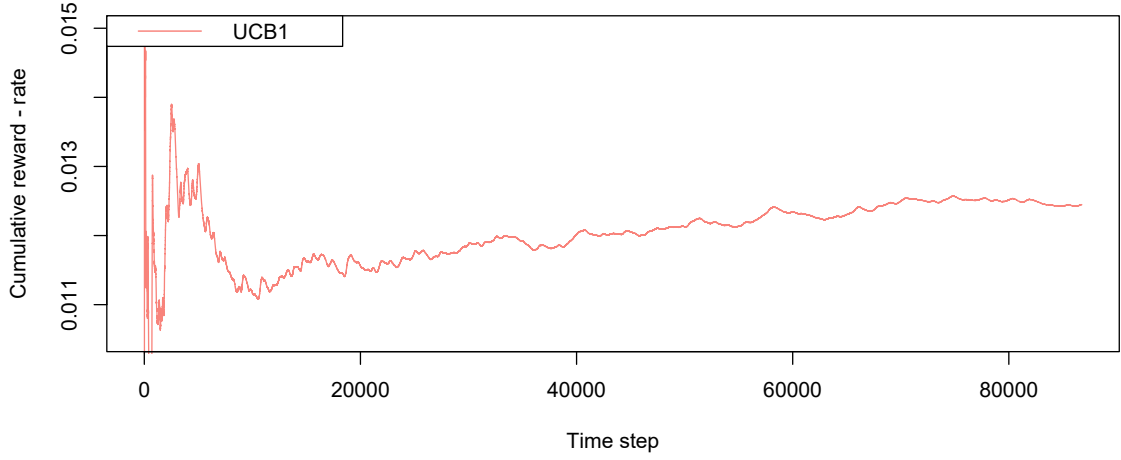
Figure 7: An UCB1 policy evaluated on a Li Bandit. The Bandit samples from 350000 rows with clicks for rewards and the display of one of four "persuasive strategies" to users of an online store representing the offline Bandit's four arms.

7. Replication of Li et al 2010

In the current section, we demonstrate how **contextual** facilitates the comparison of bandit policies on big offline datasets by running a partial replication of Li *et al.* (2010)'s seminal "A Contextual-Bandit Approach to Personalized News Article Recommendation.". The paper describes how the authors made use of offline Yahoo! click-through rate data to evaluate and compare the effectiveness of several non-contextual and contextual policies—therein introducing both the offline policy evaluator outlined in the previous section and the LinUCB algorithm introduced in section 5.4.

7.1. Description of the data

The dataset used in the Li *et al.* (2010) paper has been made available at the Yahoo! lab's website[6]). It contains the click-through rate from the Today news module on Yahoo!'s homepage over the course of several days in May 2009, totaling 45,811,883 separate, random and unbiased events.

Each row in the dataset describes an interaction event (click or no click) of users shown a randomly chosen article. Each of these events contains information on (I) six features for each of a varying subset of 19 to 25 articles shown below the the story position drawn from a pool of 217 articles by human editors (II) the id ofthe articles that is randomly chosen to be positioned at the story position at the top of the Today module on Yahoo!'s website (III) six user features, where each event contains another, distinct user (IV) information on whether the user clicked on the article at the story position, or not.

That is, for each event t an article represents one of k arms (that is, one of the 271 articles observed within the course of the 10 days covered by the dataset) with $\mathbb{R}^6$ features $X_{t,a}$ per arm, and another $\mathbb{R}^6$ features $X_{t,u}$ per unique visitor. Together, the flattened outer product of the user and article feature vector creates a $\mathbb{R}^{36}$ feature vector X_t for each user and article pair with the outcome value or reward

---

[6]At https://webscope.sandbox.yahoo.com/catalog.php?datatype=r&did=49

r_t) click (1) or no click (0). For the further details on the data structure and the general setup of the experiment, we refer the reader to Chu, Park, Beaupre, Motgi, Phadke, Chakraborty, and Zachariah (2009) and to the original Li *et al.* (2010) paper.

7.2. Data import

As the Yahoo data is too large to fit into memory, we imported most[7] of the dataset's CSV files into a MonetDB **?** instance—a fast, open source column-oriented database management system with excellent R support[8]. The import script, example import scripts for several other databases (MySQL, SQLite, Postgresql) and all other source code related to this replication can be found in the package's `demo/replication_li_2010 directory`.

7.3. Custom bandit and policies

With the Yahoo! data imported into our MonetDB server, our next step was the creation of a Bandit subclass to query the data and seven Policy subclasses implementing the policies described in the Li *et al.* (2010) paper. Though most of these Policies were already implemented in **contextual**, the fact that only a subset of all 271 articles or arms is shown to a visitor at a time meant we needed to make some minor adaptations to all of the pre-implemented policies to make our policies run smoothly on this continually shifting pool of available arms.

In effect, first, our YahooBandit receives a `self$arm_lookup` table with an overview of 271 all arms. The bandit then proceeds to look up the article id's in the lookup table at each time step *t*, and includes the indexes of the currently active arms in the context list:

```r
get_context = function(index) {
  ...
  # Retrieve the index of all arms this row/event.
  arm_indices_this_event  <- seq(10, 184, by = 7)
  article_ids             <- row[arm_indices_this_event]
  article_ids             <- article_ids[!is.na(article_ids)]
  article_ids             <- match(article_ids,self$arm_lookup)
  ...
  contextlist <- list(
    k = self$k,
    d = self$d,
    unique = self$unique, # <-- Indexes of all disjoint arms (user features)
    shared = self$shared, # <-- Indexes of all shared arms (article features)
    arms = article_ids,   # <-- Indexes of arms this event.
    X = X
  )
}
```

These indexes are then picked up by the policy classes, which use the article id's to select and update only the currently active subset of arms. For instance, in YahooEpsilonGreedyPolicy's `get_action()`:

---

[7]The first two CSV files of the Yahoo! dataset are somewhat irregular, as they contain articles with more than six features. We, therefore, decided to leave these two CSV files out of our import, resulting in 37,450,196 imported events, instead of the 45,811,883 events used in the original paper.

[8]MonetDB can be downloaded at https://www.monetdb.org/

```
  get_action = function(t, context) {
    if (runif(1) > self$epsilon) {
      # get the max of context$arms *currently in play*
      max_index            <- context$arms[max_in(theta$mean[context$arms])]
      self$action$choice   <- max_index
    } else {
      # sample from the arms *currently in play*
      self$action$choice   <- sample(context$arms, 1)
    }
    self$action
  }
```

We then initiated seven of these customized policy subclasses (Random, EGreedy, EGreedySeg, Li-
nUCB Dis, LinUCB Hyb, UCB1 and the UCB1Seg[9]) for each of the six (0, 30, 20, 10, 5 and 1
percent) levels of sparsity defined in the original paper. Resulting in $7 \times 6 = 42$ Agents, in their turn
added to one Simulation instance and run on the full dataset:

```
simulations              <- 1
horizon                  <- 37.45e6
...
con <- DBI::dbConnect(MonetDB.R(), host=monetdb_host, dbname=monetdb_dbname,
                                    user=monetdb_user, password=monetdb_password)

message(paste0("MonetDB: connection to '",dbListTables(con),"' succesful!"))

arm_lookup_table <-
  as.matrix(DBI::dbGetQuery(con, "SELECT DISTINCT article_id FROM yahoo"))

arm_lookup_table <- rev(as.vector(arm_lookup_table))

bandit <- YahooBandit$new(k = 217L, unique = c(1:6), shared = c(7:12),
                          arm_lookup = arm_lookup_table, host = monetdb_host,
                          dbname = monetdb_dbname, user = monetdb_user,
                          password = monetdb_password, buffer_size = buffer_size)

agents <-
  list (Agent$new(YahooLinUCBDisjointPolicy$new(0.2),
                  bandit, name = "LinUCB Dis",  sparse = 0.99),
        Agent$new(YahooLinUCBHybridPolicy$new(0.2),
                  bandit, name = "LinUCB Hyb",  sparse = 0.99),
        Agent$new(YahooEpsilonGreedyPolicy$new(0.3),
                  bandit, name = "EGreedy",     sparse = 0.99),
        Agent$new(YahooEpsilonGreedySegPolicy$new(0.3),
                  bandit, name = "EGreedySeg",  sparse = 0.99),
        Agent$new(YahooUCB1AlphaPolicy$new(0.4),
                  bandit, name = "UCB1",        sparse = 0.99),
        Agent$new(YahooUCB1AlphaSegPolicy$new(0.4),
                  bandit, name = "UCB1Seg",     sparse = 0.99),
        ...
        Agent$new(YahooRandomPolicy$new(),
```

---

[9]EGreedy Dis and EGreedy Hyb were too summarily described for us to be able to replicate them with confidence.

```
                    bandit, name = "Random"))

simulation <- Simulator$new(
    agents,
    simulations = simulations,
    horizon = horizon,
    do_parallel = TRUE,
    worker_max = worker_max,
    reindex = TRUE,
    progress_file = TRUE,
    include_packages = c("MonetDB.R"))

history  <- simulation$run()
...
```

7.4. Results

We were able to complete a $7 \times 6 = 42$ Agent simulation learning over all of the 37,450,196 events in our database within 22 hours on a 64 core Intel Xeon Unbuntu server with 256GB of memory. We then proceeded to use the results of the first 4.7 million events (following the original paper, representing about a day worth of events) to reproduce Li *et al.* (2010)'s Figure 4b: "CTRs in evaluation data with varying data sizes in the learning bucket.". Just as the original paper, the replicated Figure 3.2 reports each algorithm's relative CTR for all of the defined data sparsity levels, that is, each algorithm's CTR divided by the random policy's CTR.
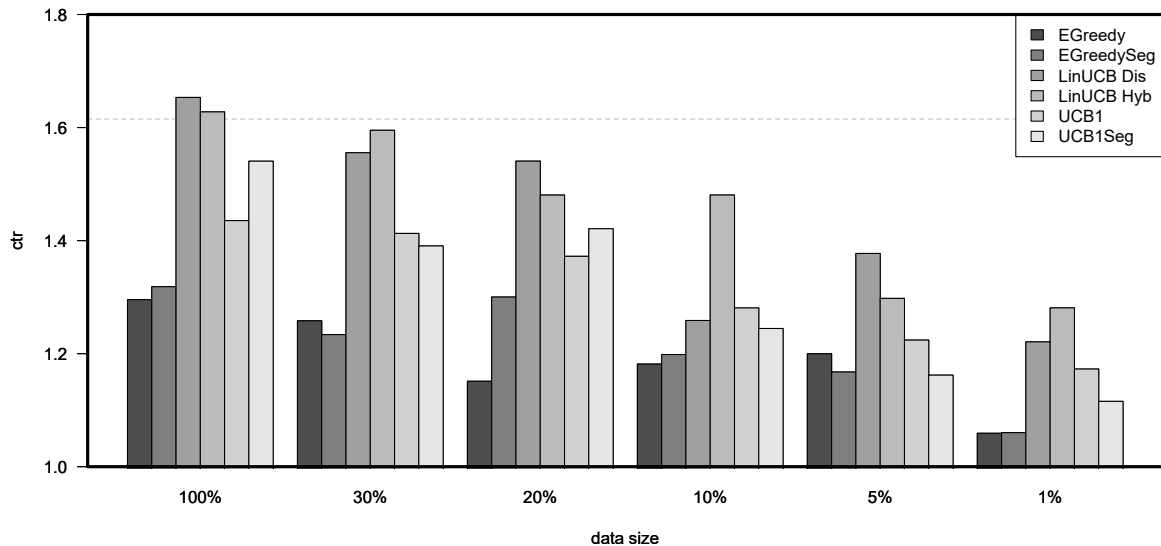

Figure 8: Replication based on the third day of data from the Yahoo! dataset of Figure 4b: "CTRs in evaluation data with varying data sizes in the learning bucket." from Li *et al.* (2010).

As can be observed in Figure 8, after one day of learning, the conclusions of the original paper still stand. First, features again prove to be of use at all levels of sparsity, as LinUCB policies outperform the others consistently. Second, UCB policies generally outperform ϵ-greedy ones. And third, Hybrid

LinUCB again shows benefits when the data is small, as can be deduced from it doing better in the 1% bucket. Still, as we started our simulation on the third day instead of the first, our results are close to, but not quite the same as those reported in the original paper. Particularly the third conclusion, that of the relative advantage of Hybrid LinUCB with sparse data, seems to be less convincing in our Figure 8.

So we decided to run a simulation that would continue to learn beyond the first day on sparse (1%) data to test whether Hybrid LinUCB's relative advantage would prove stable over time.
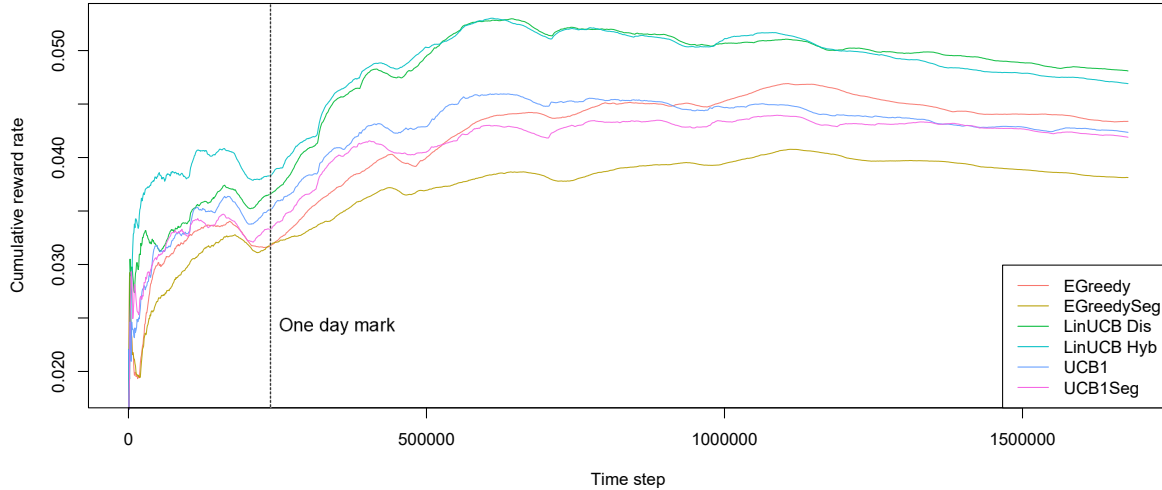
Figure 9: A plot of the cummulative reward rate (or click-through rate) for EGreedy, EGreedySeg, LinUCB Dis, LinUCB Hyb, UCB1 and UCB1Seg policies over eight days of events from the Yahoo dataset.

From this new Figure 8, it seems that after one day of training the policies did not settle yet. Looking at the full span of about eight days of learning under 1% sparse conditions, the advantage of Hybrid LinUCB over Disjoint LinUCB seems to melt away and change into an advantage of the Disjoint version. Also, surprisingly, over the full eight days, the ϵ-greedy policy goes from the worst to third best policy overall, and to best non-contextual policy—outperforming both non-contextual UCB policies. Though we intend to further analyze this discrepancy in the near future, for now, these results seem to contradict two out of three conclusions drawn from the original figure—leaving only the first outcome, the superiority of the contextual LinUCB in comparison to several non-contextual ones, standing. Underlining the benefits of **contextual**, as it enabled us to replicate and confirm the original Li *et al.* (2010) paper, and then explore it further with ease. All within about two days of work: extending Bandit and Policy classes, simulating all policy classes in parallel on an offline dataset, and plotting and analyzing the results.

8. Discussion and Future Work

Statistical computational methods, in R or otherwise, are regularly made available through single-use scripts or basic, isolated code packages (Gandrud 2016). 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 (Stodden, Guo, and Ma 2013). Such code examples offer their scientific audience a first inroad towards the comparison and further implementation of their underlying methods (Buckheit and Donoho

1995). 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 (Mesirov 2010).

Is is on that premise that we decided to develop the **contextual** R package—a package that would offer an open bandit framework with easily extensible bandit and policy libraries. To us, it made the most sense to create such a package in R (R Core Team 2018), as R is currently the de facto language for the dissemination of new statistical methods, techniques, and algorithms (Tippmann 2015)—while it is at the same time finding ever-growing adoption in industry (Muenchen 2012). 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.

In making the package publicly available, we hope to further the dissemination and evaluation of both existing and new cMAB policies. By developing the package R, we furthermore hope to offer an arena for the lively exchange of policies between scientists and practitioners—as R has proven itself to be the de facto domain specific computer language in both academia and industry, from the social and medical sciences to statistics and business analytics (Tippmann 2015).

As the package is intended to be usable by practitioners, scientists and students alike, we started our paper with a general introduction to the (contextual) multi-armed bandit problem, followed by a compact formalization. We then demonstrated how our implementation flows naturally from this formalisation, with Agents that cycle Bandits and Policies through four function calls: `get_context()`, `get_action()`, `get_reward()` and `set_reward()`. Next, we evaluated some of **contextual**'s built-in Policies, delved deeper into **contextual**'s class structure, extended **contextual**'s Bandit and Policy superclasses, demonstrated how to evaluate Policies on offline datasets, and, finally replicated a seminal cMAB paper.

Though the package is fully functional and we expect no more changes to its core architecture and API, there is ample room to further improve and extend **contextual**. We intend to expand **contextual**'s documentation and tests. We may include more Bandit paradigms, such as dueling and combinatorial bandits. We are thinking about replicating some more oft-cited cMAB papers. We expect to add a Doubly Robust Bandit (Dudík, Langford, and Li 2011). We are interested in growing our Policy library—possibly by creating a separate repository where both existing and new cMAB policies are shared, evaluated and compared. Finally, we hope, of course, that the package will find an active community of users and developers, organically growing the package in new and unexpected ways and introducing an ever-growing group of people, from business and marketing to the social and medical sciences, to the refined sequential decision strategies offered by contextual bandit policies and algorithms.

References

Abe N, Biermann AW, Long PM (2003). "Reinforcement Learning with Immediate Rewards and Linear Hypotheses." *Algorithmica*, **37**(4), 263–293. doi:10.1007/s00453-003-1038-1.

Agrawal R (1995). "The Continuum-armed Bandit Problem." *SIAM journal on control and optimization*, **33**(6), 1926–1951. doi:10.1137/s0363012992237273.

Agrawal S, Goyal N (2011). "Analysis of Thompson Sampling for the Multi-armed Bandit Problem." *arXiv*. http://arxiv.org/abs/1111.1797v3.

Agrawal S, Goyal N (2012). "Thompson Sampling for Contextual Bandits with Linear Payoffs." *arXiv*. http://arxiv.org/abs/1209.3352v4.

Auer P, Cesa-Bianchi N, Fischer P (2002). "Finite-time Analysis of the Multiarmed Bandit Problem." *Machine learning*, **47**(2-3), 235–256.

Bastani H, Bayati M (2015). "Online Decision-Making with High-Dimensional Covariates." *SSRN*.

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/.

Brezzi M, Lai TL (2002). "Optimal Learning and Experimentation in Bandit Problems." *Journal of Economic Dynamics and Control*, **27**(1), 87–108. doi:10.1016/s0165-1889(01)00028-8.

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. doi:10.1561/2200000024.

Buckheit JB, Donoho DL (1995). "Wavelab and Reproducible Research." In *Wavelets and statistics*, pp. 55–81. Springer. doi:10.1007/978-1-4612-2544-7_5.

Chang W (2017). *R6: Classes with Reference Semantics*. R package version 2.2.2, URL https://CRAN.R-project.org/package=R6.

Chu W, Park ST, Beaupre T, Motgi N, Phadke A, Chakraborty S, Zachariah J (2009). "A Case Study of Behavior-driven Conjoint Analysis on Yahoo!: Front Page Today Module." In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1097–1104. ACM. doi:10.1145/1557019.1557138.

Dudík M, Langford J, Li L (2011). "Doubly Robust Policy Evaluation and Learning." *arXiv preprint arXiv:1103.4601*.

Eckles D, Kaptein M (2014). "Thompson Sampling with the Online Bootstrap." *arXiv*. http://arxiv.org/abs/1410.4009v1.

Gandrud C (2016). *Reproducible Research with R and R Studio*. Chapman and Hall/CRC.

Kaelbling LP, Littman ML, Moore AW (1996). "Reinforcement Learning: A Survey." *Journal of artificial intelligence research*, **4**, 237–285. doi:10.1613/jair.301.

Kaptein MC, Kruijswijk JMA (2016). "Streamingbandit: A Platform for Developing Adaptive Persuasive Systems." *JSS*.

Kaptein MC, Van Emden R, Iannuzzi D (2016). "Tracking the Decoy: Maximizing the Decoy Effect through Sequential Experimentation." *Palgrave Communications*, **2**, 16082. doi:10.1057/palcomms.2016.82.

Kuleshov V, Precup D (2014). "Algorithms for ulti-armed bandit problems." *arXiv preprint arXiv:1402.6028*.

Lai TL, Robbins H (1985). "Asymptotically Efficient Adaptive Allocation Rules." *Advances in applied mathematics*, **6**(1), 4–22. `doi:10.1016/0196-8858(85)90002-8`.

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.

Li L, Chu W, Langford J, Moon T, Wang X (2012). "An Unbiased Offline Evaluation of Contextual Bandit Algorithms with Generalized Linear Models." In *Proceedings of the Workshop on On-line Trading of Exploration and Exploitation 2*, pp. 19–36.

Li L, Chu W, Langford J, Schapire RE (2010). "A Contextual-bandit Approach to Personalized News Article Recommendation." In *Proceedings of the 19th international conference on World wide web*, pp. 661–670. ACM. `doi:10.1145/1772690.1772758`.

Li L, Chu W, Langford J, Wang X (2011). "Unbiased Offline Evaluation of Contextual-bandit-based News Article Recommendation Algorithms." In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pp. 297–306. ACM, New York, NY, USA. ISBN 9781450304931. `doi:10.1145/1935826.1935878`.

Li S (2016). *The Art of Clustering Bandits.* Ph.D. thesis, Università degli Studi dell'Insubria. URL `http://insubriaspace.cineca.it/handle/10277/729`.

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.

Mandel T, Liu YE, Brunskill E, Popovic Z (2016). "Offline Evaluation of Online Reinforcement Learning Algorithms." In *AAAI*, pp. 1926–1933.

Mesirov JP (2010). "Accessible Reproducible Research." *Science*, **327**(5964), 415–416. `doi:10.1126/science.1179653`.

Muenchen RA (2012). "The Popularity of Data Science Software." URL `http://r4stats.com/articles/popularity/`.

NTUCSIE-CLLab (2018). "Striatum: Contextual Bandit in Python." URL `https://github.com/ntucllab/striatum`.

R Core Team (2018). "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. `doi:10.1214/aos/1176348382`.

Stodden V, Guo P, Ma Z (2013). "Toward Reproducible Computational Research: An Empirical Analysis of Data and Code Policy Adoption by Journals." *PloS one*, **8**(6), e67111. `doi:10.1371/journal.pone.0067111`.

Strehl A, Langford J, Li L, Kakade SM (2010). "Learning from logged implicit exploration data." In *Advances in Neural Information Processing Systems*, pp. 2217–2225.

Strehl AL, Li L, Wiewiora E, Langford J, Littman ML (2006a). "PAC model-free reinforcement learning." In *Proceedings of the 23rd international conference on Machine learning*, pp. 881–888. ACM.

Strehl AL, Mesterharm C, Littman ML, Hirsh H (2006b). "Experience-efficient Learning in Associative Bandit Problems." In *Proceedings of the 23rd international conference on Machine learning*, pp. 889–896. ACM. `doi:10.1145/1143844.1143956`.

Sutton RS, Barto AG (1998). "Reinforcement Learning: An Introduction." *IEEE Transactions on Neural Networks*, **9**(5), 1054. ISSN 1045-9227. `doi:10.1109/TNN.1998.712192`.

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. `doi:10.1007/978-3-319-51394-2_25`.

Tippmann S (2015). "Programming Tools: Adventures with R." *Nature News*, **517**(7532), 109.

Vermorel J, Mohri M (2005). "Multi-armed Bandit Algorithms and Empirical Evaluation." In *European conference on machine learning*, pp. 437–448. Springer. `doi:10.1007/11564096_42`.

Wang CC, Kulkarni SR, Poor HV (2005). "Arbitrary Side Observations in Bandit Problems." *Advances in Applied Mathematics*, **34**(4), 903–938. `doi:10.1016/j.aam.2004.10.004`.

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.

Wickham H (2014). *Advanced R*. Chapman and Hall/CRC. `doi:10.1201/b17487`.

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. `doi:10.1037/a0038199`.

Zheng R, Hua C (2016). "Stochastic Multi-armed Bandit." In *Sequential Learning and Decision-Making in Wireless Resource Management*, pp. 9–25. Springer. `doi:10.1007/978-3-319-50502-2_2`.
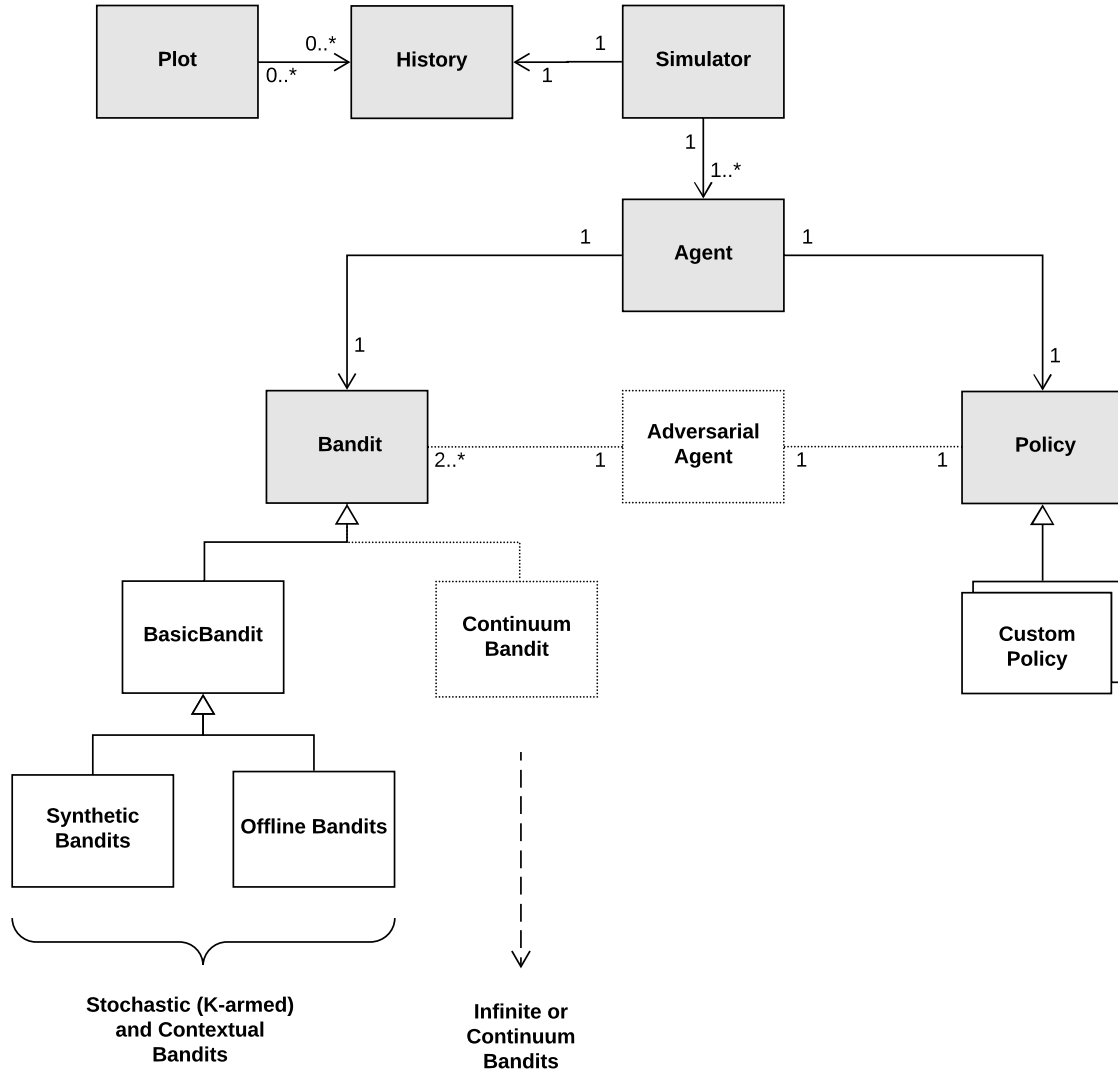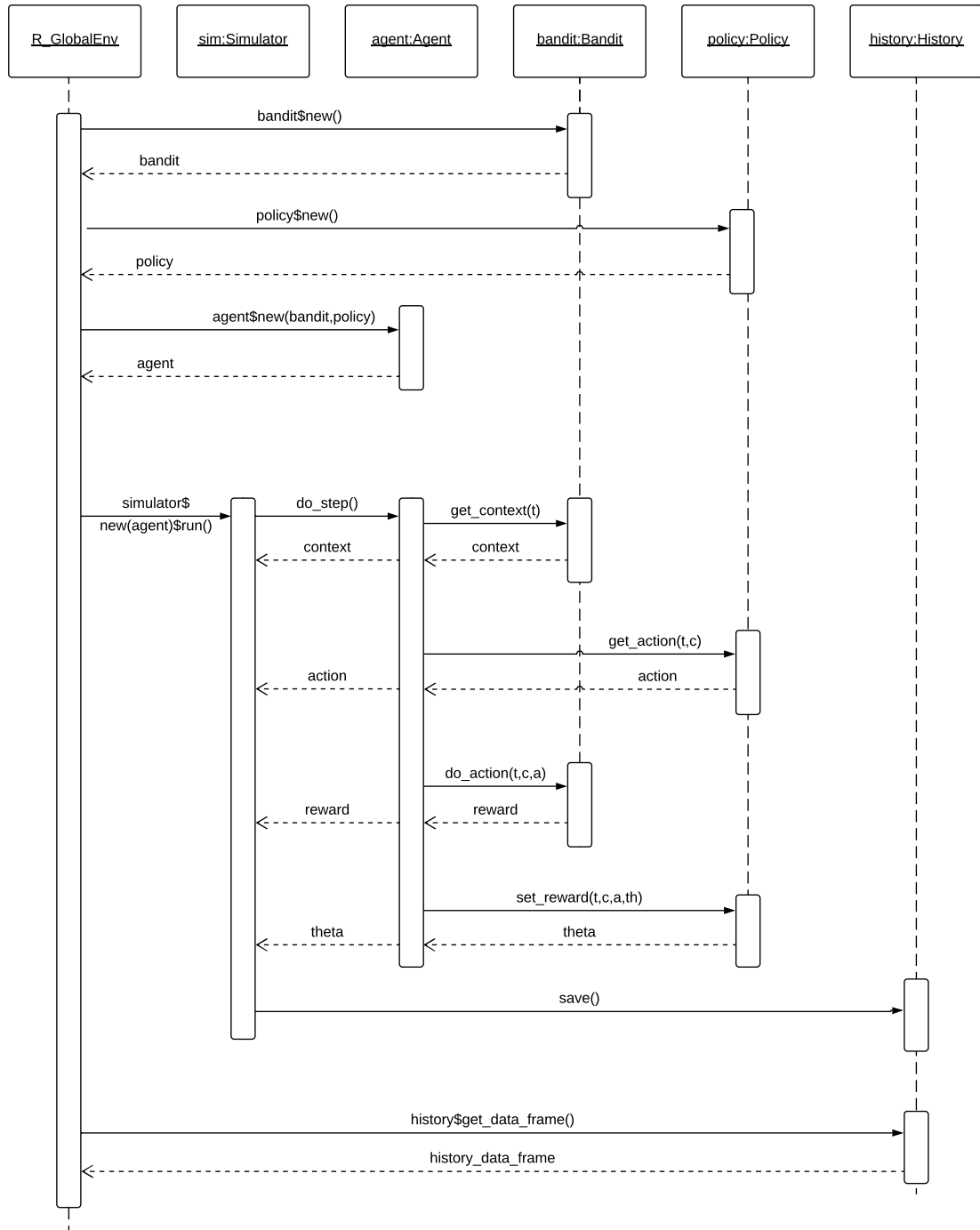
9. Appendix A: UML Diagrams

Figure 10: **contextual** UML Class Diagram

Figure 11: **contextual** UML Sequence Diagram

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