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 will receive 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 reveives 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 this formalization by adding one crucial element: contextual information. That is, cMAB policies make use of side information on the state of the world at each repeated time step t. This access to side information makes cMAB algorithms even more relevant to many real-life decision problems than its MAB progenitors: 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 a context. 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.

Still, 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 will continue with a more formal definition of MAB and CMAB problems and relate it to our implementation. In section 3, we will continue with 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 **contextual** with a policy (RVE: NOTE TO SELF: also custom bandit?) of your own. 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 formalisation to implementation

2.1. Formalisation

In formalizing the described cMAB problem, a (k-armed) **bandit** $\mathfrak{B}$ can be defined as a set of k

distributions $\mathfrak{B} = \{D_1, \ldots, D_k\}$, 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 equation 1). 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** $\mathfrak{B}$'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 often named the **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) Bandit $\mathfrak{B}$ generates reward vector $r_t = (r_{t,1}, \ldots, r_{t,k})$

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

4) Policy π observes reward r_{t,a_t} from bandit $\mathfrak{B}$ 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 miminize its cumulative regret over $t= \{1, \ldots, T\}$, defined as the sum of rewards that would have been received by always choosing optimal actions a^* over t subtracted by the sum of rewards awarded to the actually chosen actions a over t:

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

2.2. Basic Implementation

We set out to develop an implementation that stays close to the previous formalisation, while offering maximum flexibility and extendibility. As an added bonus, this kept the class structure of the package elegant and simple, with the following five classes forming the backbone of the package:

- `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 the generation of `context` vectors and `rewards` awarded to `actions` that are decided upon by its related `Policy` at each time step `t`. `Bandit` subclasses may (pre)generate these values synthetically, based on offline data, etc.

- `Policy`: The R6 class `Policy` is the parent class of all `Policies` implemented in {`contextual`}. Classes that extend this abstract `Policy` superclass superclass are expected to take into account the current `context`, together with a limited set of parameters denoted `theta` (summarizing all past actions), to chose one of a `Bandit`'s arms at each time step `t`. On receiving a `reward`, the `Policy` then uses the current `context`, `action` and `reward` to update its set of parameters `theta`.

- `Agent`: The R6 class `Agent` takes care of the flow of information between and the running of one `Bandit`/`Policy` pair.

- `Simulation`: The R6 class `Simulation` is the entry point of any **contextual** simulation. It encapsulates one or more `Agents`, clones them if necessary, runs the `Agents`, and saves the log of all of the `Agents` interactions to a `History` object.

- `History`: encapsulates an R `data.table`, keeps a log of all interactions, and provides data summaries.

- `Plot`: The R6 class `Plot` generates plots based on `History` data. It can also be invoked by calling the generic `plot(h)`, where h is an `History` class instance.

From this follows that in our framework, on being run by the Simulator, an Agent repeatedly performs the following operations at every t in $t=1,2,\ldots,T$ (see also Figure 2.2)

1a) `Agent` asks the `Bandit` for side information that might influence the expression of its arms by calling `Bandit$get_context()`

1b) `Bandit` returns feature vector *Xt*.

2a) `Agent` asks `Policy` π which of the `Bandit`'s k arms to choose given *Xt* by calling `Policy$get_advice(Xt)`

2b) `Policy` π returns action a_t based on the state of its parameters θ_t.

3a) `Agent` does action a_t by playing the suggested `Bandit` arm through a call to `Bandit$do_action(Xt, at)`

3b) `Bandit` rewards the `Agent` by returning reward r_t for action a_t.

4a) The `Agent` sends the reward r_t to `Policy` π by calling `Policy$set_reward(Xt,at,rt))`

4b) `Policy` π uses r_t to update the `Policy`'s set of parameters θ_t given *Xt*.

Where we assume that at each time step t, all information necessary to choose an arm is summarized using a limited set of parameters denoted θ_t, whose dimensionality is much smaller than of the log of all historical interactions.
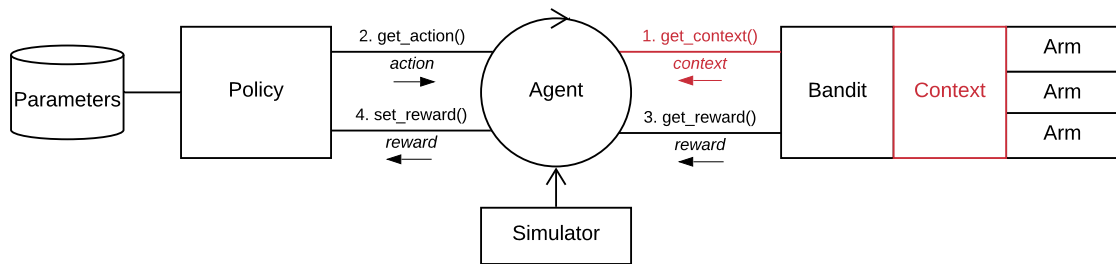

Figure 1: Basic model of the logical flow of an **contextual**. The extension from MAB towards cMAB models is higlighted in red.

3. Object-oriented setup of the package

3.1. 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. But when a set of well-researched interrelated algorithms, such as MAB and cMAB policies, find growing academic, practical and commercial adoption, it becomes crucial to offer a more standardized and more accessible way to compare such methods and algorithms.

It is against this background that we decided to develop the **contextual** R package–a package that offers an easily extendible and open bandit framework together with an extensible bandit library that presents some of the most well 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.

Given this background, it was of critical importance to make our R based framework as concise and as easily extensible as possible. For us, this meant it made the most sense to build our package on the relatively new R6 object system, as the R6 system allowed us to create readily intelligible, coherent classes that made use of R6's reference semantics. These reference semantics are similar to R's built-in reference classes. But compared to these classes, the resulting R6 classes are simpler and lighter-weight, and (as they are not built on S4 classes) do not require the methods package. This made **contextual** substantially less resource-hungry and its resulting objects much smaller than they would otherwise have been–of great importance in a simulation package such as **contextual**.

Another, more general advantage of making use of an object system such as the R6 system is that such systems support class inheritance. In other words, we were able to create subclasses (for example, ContextualBandit) that can add or override methods to their superclasses (for example, Bandit). This enabled us to translate the cMAB formalization from section 2 almost one on one to a transparent object-oriented structure.

To clarify the resulting class structure, we have created 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 X 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 X, 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 some of **contextual**'s main objects in action.

3.2. Main classes

At least policy, and often the bandit, ...Two of the classes more in depth, as those are the ones that generally need to be extended for the evaluation of Policies. Let's now take a closer look at both UML diagrams, and go over each of our classes one by one.

Bandit

The abstract class `Bandit` is the super class 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()`:

```r
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$d and, where applicable, context vector X.
      list(k = n_arms, d = n_features, X = context)
    },
    do_action = function(action, t) {
      stop("Bandit subclass needs to implement bandit$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.",
    }
  )
)
```

Where possible, it is advisable to pregenerate or precache Bandit contexts and rewards, as this is computationally much more efficient than the repeated generation of these vectors. To facilitate this, during initialisation **contextual** calls `generate_bandit_data()` for every Bandit where `self$precaching` is `TRUE`.

contextual makes several Bandits available out of the box. For each Bandit, there is at least one example script, to be found in the package's `demo` directory. The currently available Bandits are:

- `BasicBandit`: this basic k-armed bandit synthetically generates rewards based on a weigh vector that has to set at instantiation. It does not return context vector X.

- `ContextualBandit`: a basic contextual bandit synthetically that 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 vectors.

- `LiBandit`: a basic example of a bandit that makes use of offline data - here, an implementation of Li(2232)

- `OfflineBandit`: an example of a more complex offline bandit, that applies the doubly robust estimation technique to policy evaluation

These prefab bandits can be used to test policies without further adue. But 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

Next to `Bandit`, the second crucial contexual superclass is `Policy`. Just like `Bandit`, this abstract class also declares methods without offering an implementation - but here, it is all `Policy` subclasses that have to implement them. Specifically, any policy, an therefor every Policy subclass, has to implement at least `set_parameters()`, and, particularly, `get_action()` and `set_reward()`:

```r
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$action <- list()  # initiatlisation of action list for internal use
    },
    get_action = function(context, t) {
      # chooses arm based on self$theta and context, returns its index in action$choice
      stop("Policy$get_action() has not been implemented.", call. = FALSE)
    },
    set_reward = function(context, action, reward, t) {
      # 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
    }
  )
)
```

Agent

To ease the encapsulation of isolated (parallel) Bandit and Policy simulations, an Agent class takes one Bandit subclass and one Policy subclass as its arguments:

```
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 binds Bandit and Policy classes together, keeps track of time through its private variable state$t, and makes 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(
    #...
    step = function() {
      private$state$t <- private$state$t + 1
      list(context = bandit_get_context(),
           action  = policy_get_advice(),
           reward  = bandit_do_action(),
           theta   = policy_set_reward())
    }
    #...
  )
)
```

Simulation

An instance of a Simulator class takes (at least) an Agent or a list of Agents, the horizon, and number of simulations to complete a basic contextual simulation setup. When run, by calling simulator_instance$run(), its starts cloning and then running Agents, potentially dividing the agents over different parallel nodes. It then receives and accumulates all results, and saves these into an History object:

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

History

History objects aggregates the data acquired during a Simulation in its private data.table log. You can plot() a History object, summarize() it, or, amongst others, obtain either a data.frame() or a data.table() of said log:

```
history_dt <- history$get_data_table()
```

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 common, by calling plot() with a History instance plus plot type for arguments.

```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)
```

4. Basic use of the package

Here, we show how to simulate some bandits, with their current implementation.

4.1. Epsilon First

In this 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 may be a 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.

For figures, see Figure 2 on page 11.

The policy:

The EpsilonFirstPolicy class:

```r
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)
```

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

```
    },
    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
    }
  )
)
```

Running the policy:

```
library("contextual")

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

policy               <- EpsilonFirstPolicy$new(first = 50, name = "EFirst")
```

```r
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)
```
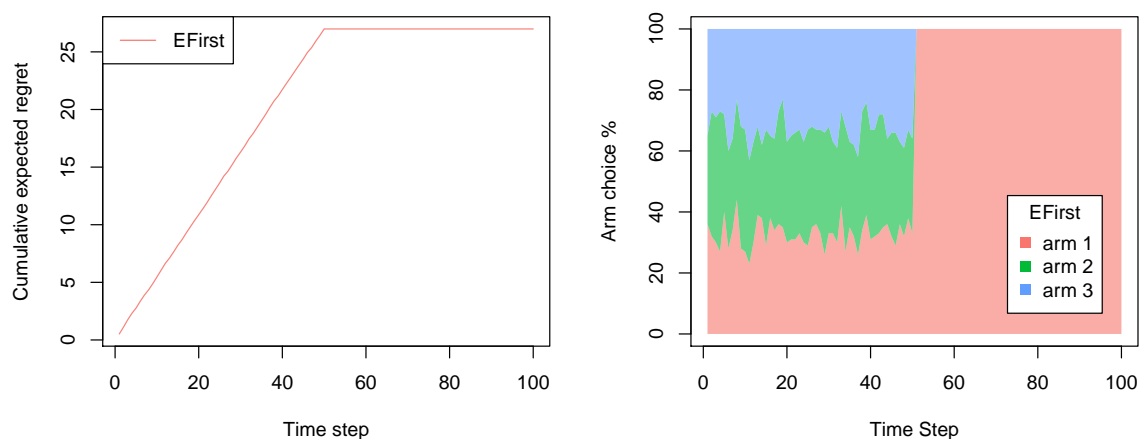

Figure 2: Epsilon First

4.2. Epsilon Greedy

This is an algorithm for continuously balancing exploration with exploitation. A randomly chosen arm is pulled a fraction ϵ of the time. The other 1-ϵ of the time, the arm with highest known payout is pulled.

For figures, see Figure 3 on page 13.

The algorithm:

Translated to the EpsilonGreedyPolicy class:

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

```r
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
    }
  )
)
```

How to run it:

```r
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)
```
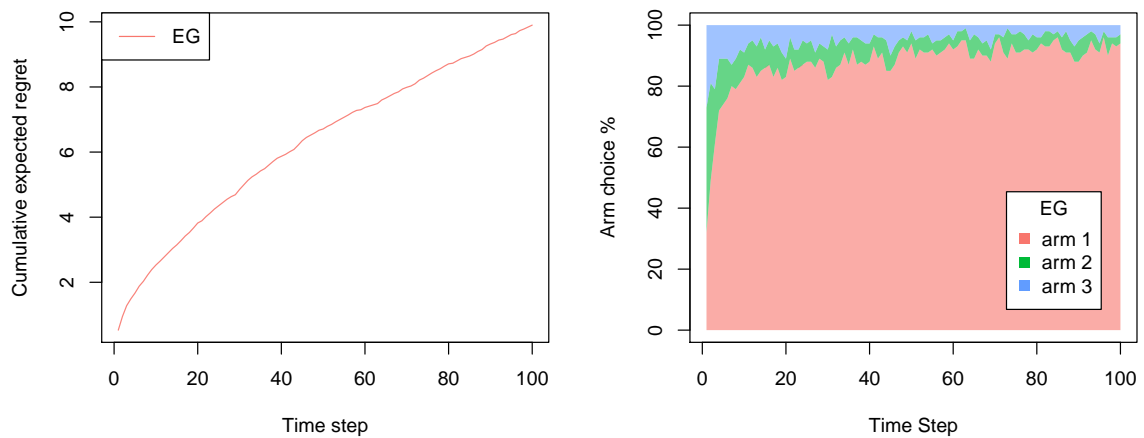

Figure 3: Epsilon Greedy

4.3. Contextual Bandit: LinUCB with Linear Disjoint Models

The algorithm:

This is how the algorithm works: at each step, we run a linear regression with the data we have collected so far such that we have a coefficient for each context feature. We then observe our new

---

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

---

context, and generate a predicted payoff using our model. We also generate a confidence interval for that predicted payoff for each of the three arms. We then choose the arm with the highest upper confidence bound.

For figures, see Figure 4 on page 16.

```r
#' @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
    },
```

```r
    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
    }
  )
)
```

```r
horizon           <- 100L
simulations       <- 300L
                            # k=1   k=2   k=3          columns represent arms
weights  <- matrix(         c( 0.9,  0.1,  0.1,   # d=1
                               0.1,  0.9,  0.1,   # d=2
                               0.1,  0.1,  0.9),  # d=3   rows for context features

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

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

agents            <- list( Agent$new(EpsilonGreedyPolicy$new(0.1, "EpsilonGreedy"), 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", grid = TRUE)
plot(history, type = "cumulative", regret = FALSE, grid = TRUE)
```
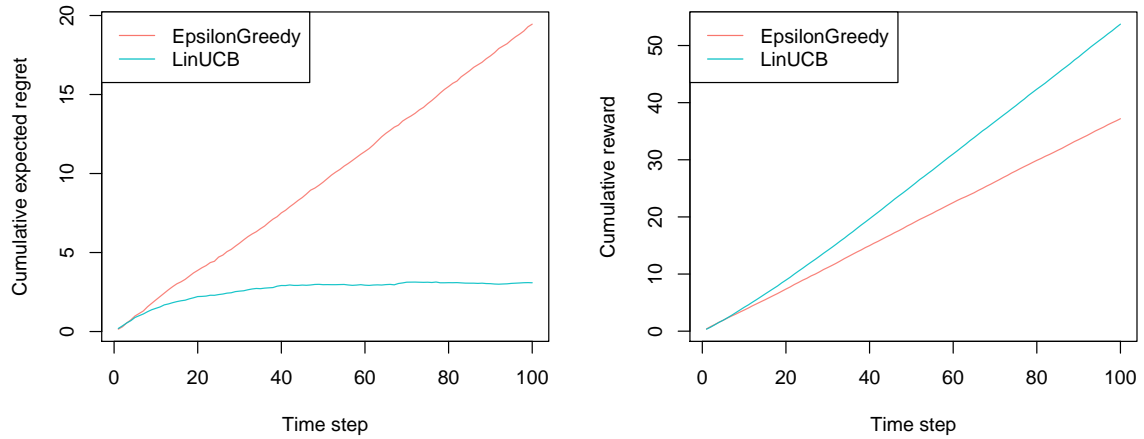
Figure 4: LinUCB algorithm with linear disjoint models, following Li et al. (2010)

5. Extending the package

Through its R6 based object system, it's relatively easy to extend **contextual**. Below, we demonstrate how to make use of that extensibility through the implementation of a PoissonRewardBandit extending **contextual**'s BasicBandit class, and of an PoissonRewardBandit version of the Epsilon Greedy policy presented above.

```r
PoissonRewardBandit <- R6::R6Class(
  "PoissonRewardBandit",
  # Class extends BasicBandit
  inherit = BasicBandit,
  public = list(
    initialize = function(weights) {
      super$initialize(weights)
    },
    # Overrides BasicBandit's do_action to generate Poisson based rewards
    do_action = function(context, action, t) {
      reward_means = c(2,2,2)
      private$R <- matrix(rpois(3, reward_means) < 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
```

```r
  inherit = EpsilonGreedyPolicy,
  portable = FALSE,
  class = FALSE,
  public = list(
    # Override get_action, use annealing epsilon
    get_action = function(context, t) {
      self$epsilon <- 1 / log(t + 0.0000001)
      super$get_action(context, t)
    }
  )
)

weights     <- c(7,1,2)
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 = 200L, simulations = 100L, 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)
```
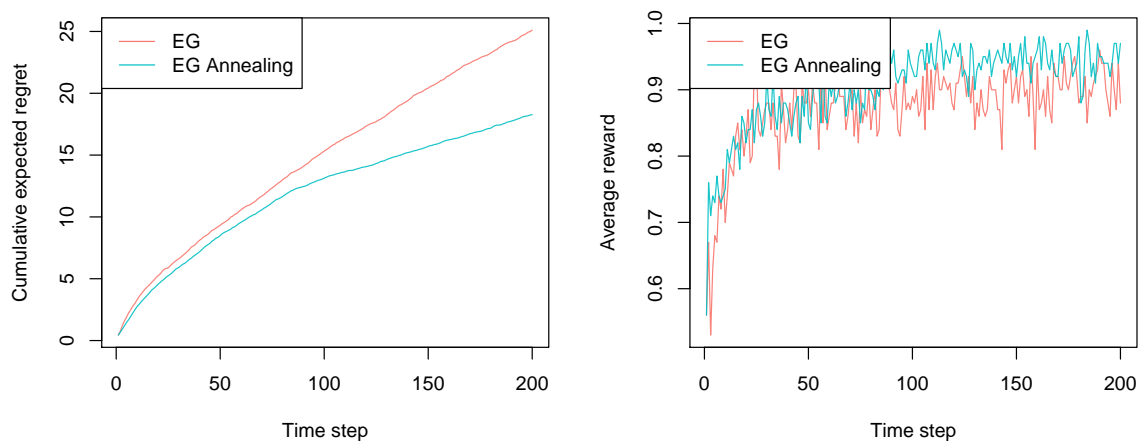

Figure 5: Extending BasicBandit and EpsilonGreedyPolicy

6. Simulation and Offline evaluation Bandits

6.1. Simulation

Some info on the implemented simulating Bandits, inc strengths and weakenesses.

*** Basic very simple ***

*** Based on modeling ***

6.2. Offline evaluation

Offline evaluation through LiLogBandit

Though it is, as demonstrated in the previous section, relatively easy to create basic simulators to test simple MAB and cMAB policies, the creation of more complex 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 complex than the analysis and implementation of the policies themselves. More seriously, even when succeeding in surpassing these technical challenges, it remains an open question if an evaluation based on simulated data reflects real-world applications, as modeling by definition introduces bias.

But there exists another, unbiased approach to testing MAB and cMAB policies. This approach makes use of widely available offline sources of data and can pre-empt the issues of bias and model complexity. It also offers the secondary advantages that offline data is both widely available and reflective of real-world online interactions. But there is one catch, that is particular to the evaluation of MAB problems: when we seek to make use of offline data, we miss out on user feedback when a policy advices a different arm than the one the user selected. In other words, offline data is only "partially labeled" with respect to any Bandit policies, as bandit evaluations only contain user feedback for arms that were displayed to the agent but include no information on other arms.

*** explain how li log algorithm helps here***
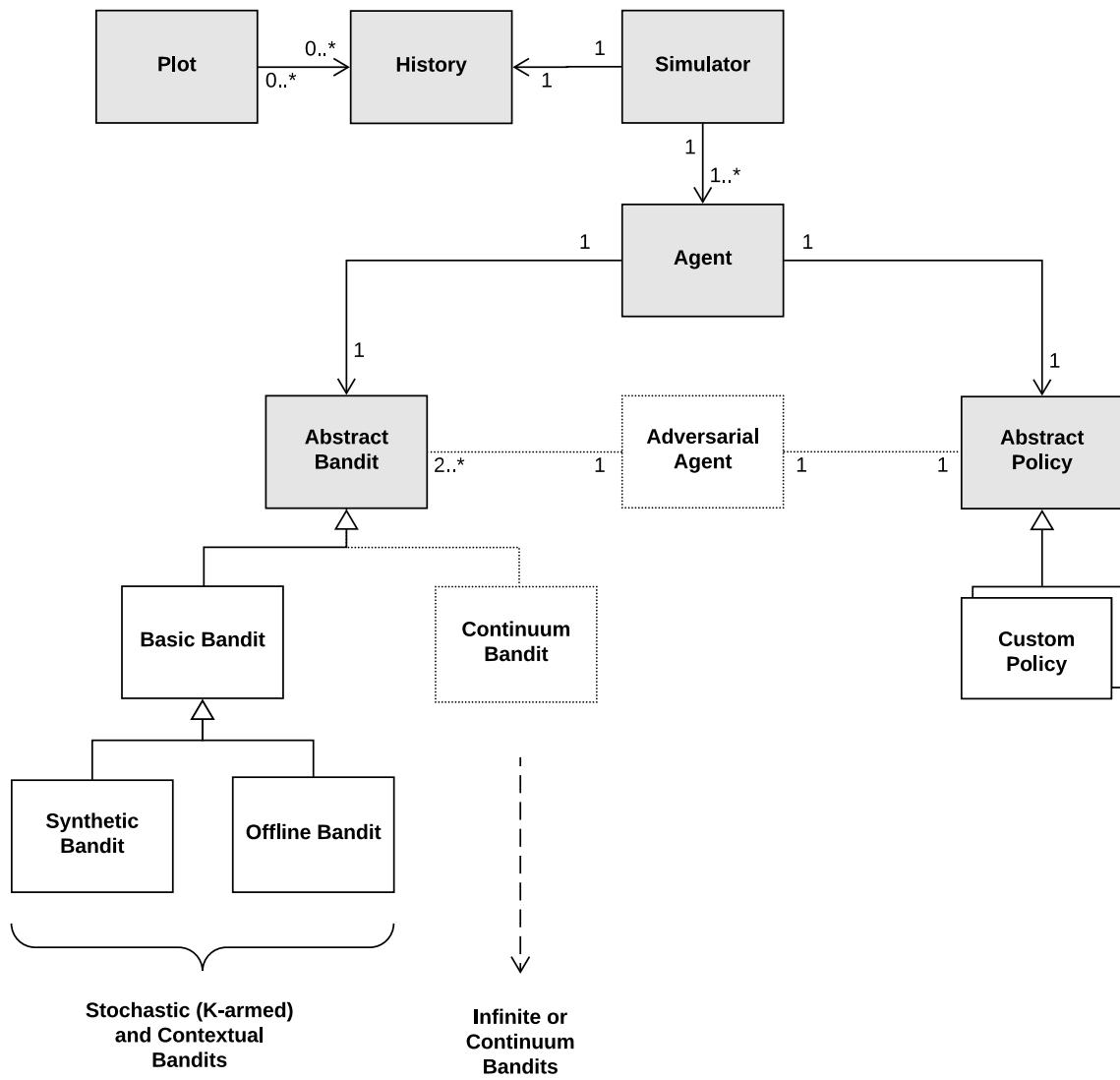
*** insert algorithm ***

*** insert code ***

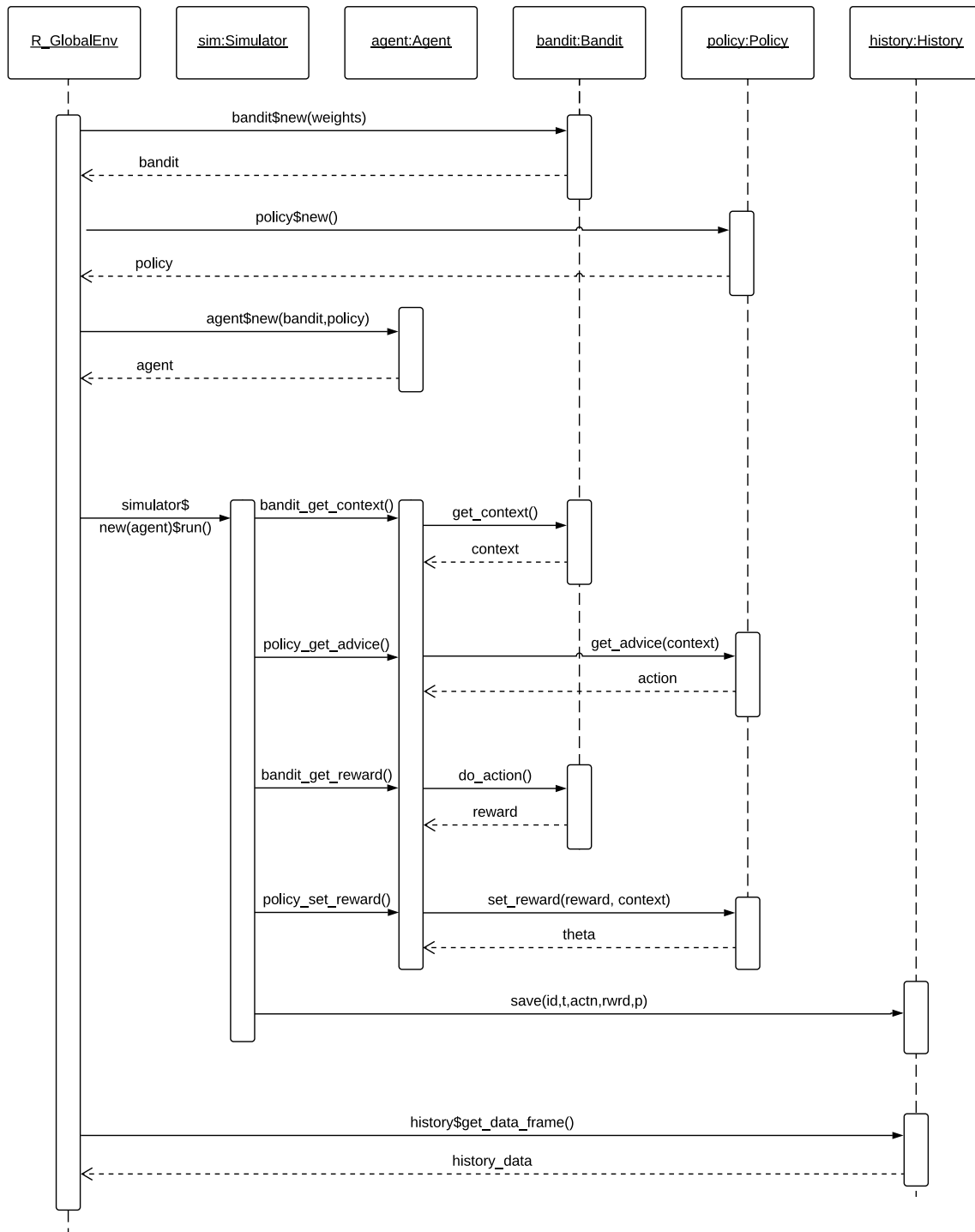*Offline evaluation through DoublyRobustBandit*

*** insert algorithm ***

*** insert code ***

7. Replications with offline data

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

Figure 6: **contextual** UML Class Diagram

Figure 7: **contextual** UML Sequence Diagram

8. Special features

For instance, quantifying variance..

9. 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

10. Extra greedy UCB

Ladila bladibla.

11. 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.

12. 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