Experimenting with sequential allocation procedures

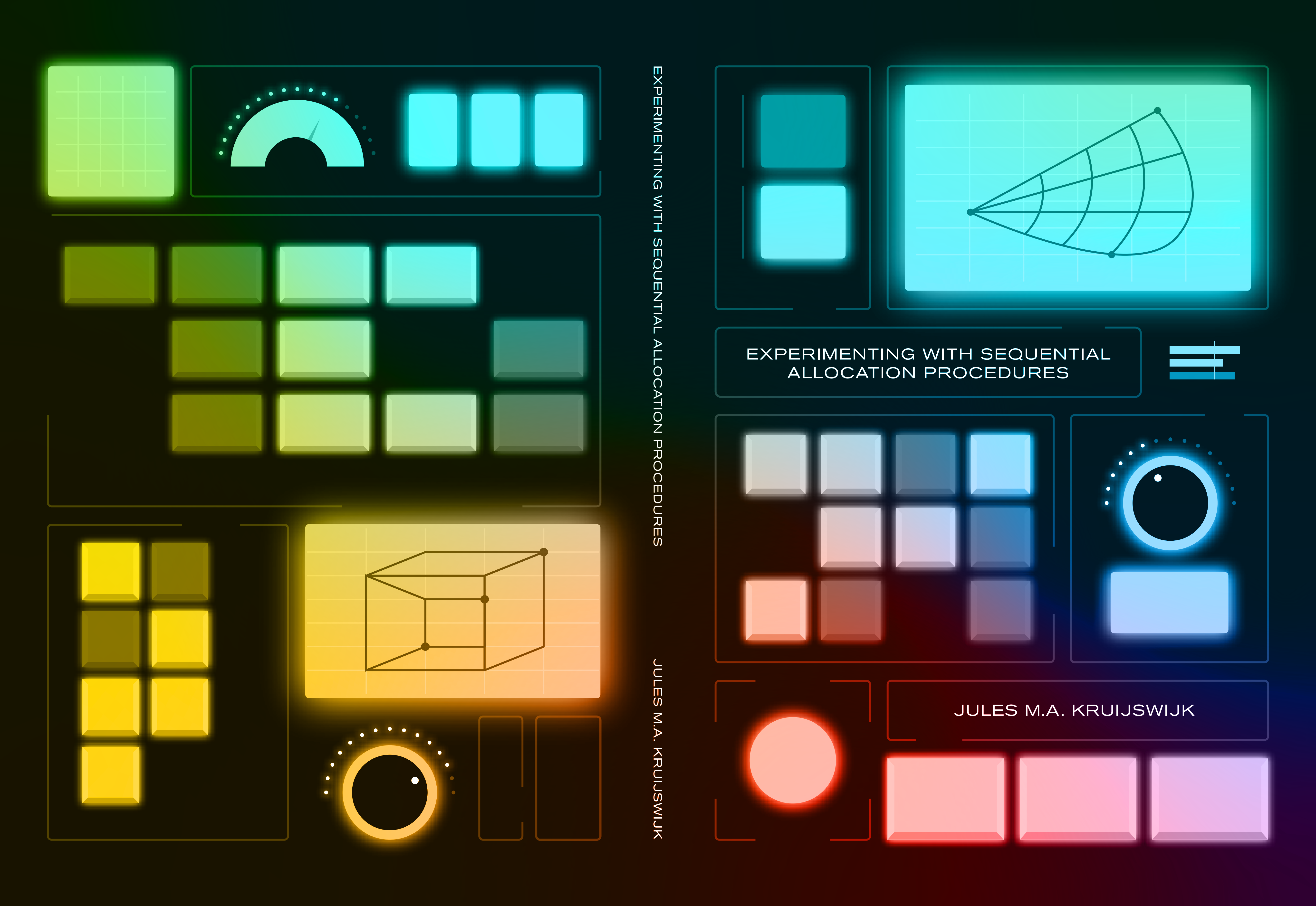
Kruijswijk, J.M.A.

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Link to publication in Tilburg University Research Portal

Citation for published version (APA):
Kruijswijk, J. M. A. (2021). *Experimenting with sequential allocation procedures*. Print.com.



EXPERIMENTING WITH SEQUENTIAL ALLOCATION PROCEDURES

JULES M.A. KRUIJSWIJK

EXPERIMENTING WITH SEQUENTIAL ALLOCATION PROCEDURES

JULES M.A. KRUIJSWIJK

EXPERIMENTING WITH SEQUENTIAL ALLOCATION PROCEDURES

JULES M.A. KRUIJSWIJK

Colophon

Copyright original content © 2020 Jules M. A. Kruijswijk, CC-BY 4.0

Printing was financially supported by Tilburg University.

Printed by:	Print.com
Cover design:	Lars Posthumus, smallmediumlars - www.smallmediumlars.nl

Experimenting with Sequential Allocation Procedures

Proefschrift ter verkrijging van de
graad van doctor aan Tilburg University
op gezag van de rector magnificus,
prof. dr. W. B. H. J. van de Donk,

in het openbaar te verdedigen
ten overstaan van een door het college
voor promoties aangewezen commissie
in de Aula van de Universiteit op
vrijdag 19 februari 2021 om 13:30 uur

door

Jules Maxim Alexandre Kruijswijk

geboren te Hilvarenbeek

Promotores:	prof. dr. J. K. Vermunt (Tilburg University)
	prof. dr. M. C. Kaptein PDEng (Tilburg University)
Promotiecommissie:	prof. dr. L. G. M. T. Keijsers (Erasmus University Rotterdam)
	prof. dr. E. Marchiori (Radboud University Nijmegen)
	prof. dr. E. Postma (Tilburg University)
	prof. dr. ir. A. P. de Vries (Radboud University Nijmegen)

Table of Contents

Table of Contents

Chapter 1

Introduction

1.1 Sequential allocation procedures

Many disciplines within science use experiments to test and validate theoretical ideas and expectations (for an example, see Kaptein, De Ruyter, Markopoulos, & Aarts, 2012). In most experiments done in the social and behavioral sciences, each subject, participant or user is allocated to a single treatment condition chosen from a set of possible treatments. An integral piece of the design of an experiment concerns the decision on how to allocate subjects to the various treatments – in other words, the decision on the treatment allocation procedure. In the classical experiment, the allocation is random and, moreover, the whole treatment allocation is performed before the actual experiment starts (Fisher, 1990). The treatment allocation procedure can thus be defined as follows: randomly assign each of the study subjects to one of the treatment conditions. Once the outcome of the experiment is recorded for all subjects, the resulting data set can be used to conduct the analysis specified beforehand (e.g. comparing means across treatment conditions). Figure 1.1 visualizes the data collection in a classical experiment.

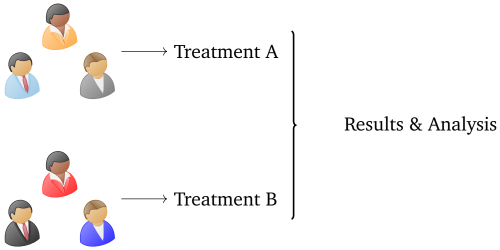


Figure 1.1: The traditional way of collecting data for an experiment. Subjects are allocated to treatment conditions a priori and the collected data is analyzed only after all treatment assignments have been carried out.

However, in most experiments, participants do not arrive all at the same time but instead one-by-one. In such a situation, the experiment concerned is, in fact, conducted in a sequential manner, where subjects are allocated to a treatment when, or just before, they arrive. This means that, alternatively to the classical experiment discussed above, we can view the data generating process as a dynamic process; that is, we have constant interactions with subjects, where each interaction concerns choosing a treatment and observing an outcome for a single subject. As a consequence, at the start of an interaction with a new subject, we often have the data available from all previous interactions (see Figure 1.2 for a visual overview), implying these data can potentially be used to influence the decision on the treatment allocation of the subject concerned. Thus, treatment allocation can be carried out sequentially, potentially benefitting from the data generated in previous interactions.

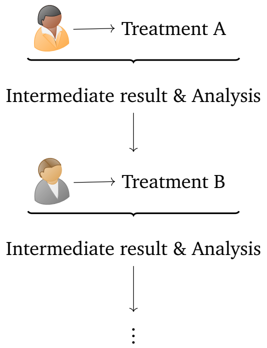

Treatment A
Intermediate result & Analysis
Treatment B
Intermediate result & Analysis

Figure 1.2: A sequential view on collecting data for an experiment. Subjects are allocated to treatment conditions one-by-one and the data resulting from previous subjects could potentially affect the treatment allocation of the current subject.

The sequential allocation of treatments is largely researched under the heading of the multi-armed bandit problem (see e.g., Berry & Fristedt, 1985; Gittins, 1989; Scott, 2010; Whittle, 1980). The name and the original problem originate from the following example: suppose we face a number of slot machines (some of which are named one-armed bandits), each with a potentially different payoff. It is our goal to make as much profit (or, in the case of gambling, as little loss) as possible by sequentially choosing which machine to play, while learning from the observations as we go along (Berry & Fristedt, 1985; Whittle, 1980). We face a trade-off between exploration and exploitation (Macready & Wolpert, 1998): on the one hand, we wish to play the machine that was successful in earlier attempts as often as possible (exploitation), but on the other hand, we wish to find the machine with the highest payoff through experimentation (exploration). The MAB problem, and its generalization, the contextual MAB (or CMAB) problem – in which before selecting a machine we observe the state of the world that could be related to the optimal choice of machine at that point in time – yields a flexible formalization for studying sequential treatment allocation procedures in the social sciences and beyond (Eckles & Kaptein, 2019), and many solutions are povided that are researched both empirically as well as theoretically (see e.g., Agrawal & Goyal, 2013b; Dudík, Hsu, et al., 2011; Lattimore & Szepesvári, 2020; Li, Chu, Langford, & Wang, 2011). Interestingely, the traditional experiment – and the subsequent treatment decision that we might make – can be regarded as one possible solution to the MAB problem: we a priori decide about a period of uniform random exploration on the machines (i.e., the number of subjects we randomly assign to the treatments), after which we play the machine with the highest average reward (i.e., we assign all remaining subjects to the best treatment).

More formally, the CMAB problem can be defined as follows: at each time $t = 1, ..., T$, we observe a context $x_t \in \mathcal{X}$. After we choose an action (or treatment) $a_t \in \mathcal{A}$, we observe reward r_t from an unknown probability distribution $P(r|a, x)$. The aim is to find a *policy* Π – which is a mapping from all the historical data $\mathcal{D}$ (containing all previous $(x_1, a_1, r_1), \ldots, (x_{t-1}, a_{t-1}, r_{t-1})$ triplets) and the current context x_t to the next action (a_t) – that selects actions such that the cumulative reward $R_c = \sum_{t=1}^{T} r_t$ is as large as possible. The canonical MAB problem can be considered as a CMAB problem but with empty context x_t. Figure 1.3 shows how the CMAB formalization relates to Figure 1.2.

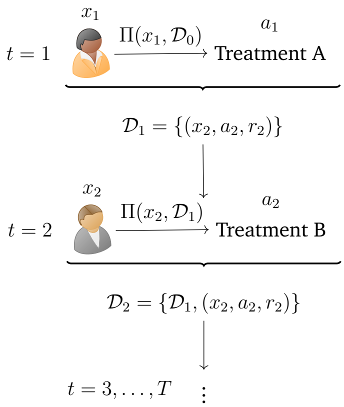

x_1
$t = 1$
$\Pi(x_1, \mathcal{D}_0)$
a_1
Treatment A
$\mathcal{D}_1 = \{(x_2, a_2, r_2)\}$
x_2
$t = 2$
$\Pi(x_2, \mathcal{D}_1)$
a_2
Treatment B
$\mathcal{D}_2 = \{\mathcal{D}_1, (x_2, a_2, r_2)\}$
$t = 3, \ldots, T$
$\vdots$

Figure 1.3: A sequential view on collecting data for an experiment using formal notation that is used throughout the thesis.

The aim of this thesis is to introduce experimentation using sequential allocation procedures to social scientists, and, moreover, to develop new methods for dealing with specific problems associated with social science applications. Firstly, we introduce a flexible framework and a software tool that allow for easy implementation and deployment of experiments, either using traditional or sequential allocation procedures, in (web-based) research: Chapter 2 provides details about the general framework and the software tool and Chapter 5 illustrates how to integrate the software in web-based research. Notably, in these chapters, we introduce a novel approach for specifying MAB policies with *getAction* and *setReward* operations (or *decision* and *summary* steps), a useful formalization that since has been used by various authors (Agarwal et al., 2017; van Emden & Kaptein, 2018). Secondly, new methods are developed for designing MAB policies that are explicitly valuable for social science applications: Chapter 3 introduces a method for dealing with designs yielding data sets with dependent or nested observations and Chapter 4 develops a method for performing offline evaluations with continuous treatments (more details are provided below). In both cases, our new approaches are inspired by empirical MAB policy applications in the social sciences, where both nested observations and continuous treatments are common, while such issues are not commonly addressed in the mostly theoretical MAB literature. We thus also contribute to the MAB literature by introducing new problem formalizations and by empirically examining our proposed solutions.

In Section 1.3, more details are provided on the content of the separate chapters using a motivating example, which will first be presented in Section 1.2. Note that the four main chapters are submitted or published as separate journal articles, which may lead to some overlap, repetition, and possibly also inconsistencies in notation across the chapters. We, however, chose to keep the chapters as similar as possible to their original sources.

1.2 A motivating example

Multiple studies have shown positive effects of so-called internet-based behavioral interventions, be it for improving mental health (Baumeister, Reichler, Munzinger, & Lin, 2014), decreasing addictive behavior (Chebli, Blaszczynski, & Gainsbury, 2016), or promoting physical health (van den Berg, Schoones, & Vlieland, 2007). Kaptein et al. (2012) showed that by tailoring text messages to the personal susceptibility to specific social influence strategies, a larger decrease of snacking consumption was achieved. While their experiment tailored the text messages based on information obtained with a questionnaire, such experiments could also, as indicated by Kaptein, Markopoulos, De Ruyter, and Aarts (2015), benefit from treatment personalization (in this case the context of the text messages) using background characteristics of the subject concerned, as well as results obtained with earlier subjects, thus giving rise to a contextual bandit problem (see also Tewari & Murphy, 2017).

As an example application, imagine a study aimed at stimulating the physical activity of subjects by influencing the amount of steps they walk during a day. Subjects wear an activity monitor (e.g., a smartwatch) measuring the number of steps and, at the beginning of each day, receive information on a goal regarding the number of steps for the day concerned. This goal is the treatment a_t in terms of the MAB formalization, which in this case is modeled as a continuous treatment, while the amount of steps taken on a specific day is the reward r_t. Note that in this experiment the researchers interact with the same subjects over a longer period of time, which results in a data set with a nested structure.

We assume that within our example experiment, the objective is to try to maximize the realized daily number of steps by personalizing the goals. The optimal choice for the daily goal can potentially be influenced by the context x_t, which contains subject characteristics, such as age, gender, and weight. Note that the interest is in improving the health of the subjects as much as possible rather than making a deterministic choice on the amount of steps (i.e., which suggested amount of steps is a better overall treatment). In other words, it is assumed that obtaining an as high as possible cumulative reward R_c (i.e., in total as many steps as possible) will result in overall maximum health. Thus, while sequentially interacting with subjects over multiple days, the overall aim of the experiment is to choose the amount of daily steps per subject maximizing the realized number of steps taken by this subject during this day, and thereby maximizing the total number of steps across subjects and days. Finding a good policy that will maximize the total number of staps taken might itself also be considered the target of the experiment.

1.3 Outline

To address the problem of finding a good policy that ensures that people walk more (as sketched in the example application described above), we can rely on a large body of mainly theoretical literature on sequential allocation procedures for conducting experiments with a continuous treatment range (for examples, see Agrawal, 1995; Kaptein & Ianuzzi, 2016; Kleinberg, 2004). However, practical implementation and evaluation of the proposed treatment allocation procedures in (online) field experiments remains difficult. For instance, for our example experiment, we need software that allows communicating with a smartwatch such that each day a new goal can be set for the subject concerned. Moreover, we would most probably wish to use a set of tools making it easy to modify the desired behavior of the treatment allocation procedure once the experiment has been set up. Chapter 2 introduces **StreamingBandit**, a Python web application for developing and testing allocation procedures in field studies, and moreover introduces a flexible framework identifying MAB policies to consist of two steps: a *summary* step (where we update the parameters of the policy) and a *decision* step (where we use the policy to choose a treatment). The software allows experimenters to sequentially select treatments in real time, but also to quickly develop and re-use novel allocation procedures. The framework constraints the implemented procedures to be computationally efficient by restricting them to row-by-row updating (Michalak, DuBois, DuBois, Wiel, & Hogden, 2012) – a feature that is desirable in many practical applications (Agarwal et al., 2017; Ansari & Mela, 2003). Also, once **StreamingBandit** is integrated in a particular experiment, it allows modifying the treatment allocation procedure with just a few lines of code – thus allowing for easy experimentation with multiple treatment allocation procedures within field studies. The chapter details the complete implementation logic of **StreamingBandit**, and gives numerous examples illustrating the usefulness and flexibility of the software. In our example experiment, **StreamingBandit** will allow communicating with our subjects' smartwatches, which can ask a server running **StreamingBandit** the goal for the subject concerned. The treatment allocation procedure implemented in **StreamingBandit** computes the personalized goal (the proposed number of steps), and returns this information to the subject's smartwatch.

In the example experiment, we wish to use a treatment allocation procedure to personalize the daily goal for each subject. For this purpose, we certainly want to use the historical data as efficiently as possible. Before giving any recommendation to a new subject, we can use the data from all previous subjects to compute an average goal that is neither too high or too low. However, as more observations (days) from of a subject comes in, we can start using the subject-specific data to compute a completely personalized goal. In this case, we consider the data to be (hierarchically) dependent: observations are nested within subjects over time (Gelman & Hill, 2006; Ippel, Kaptein, & Vermunt, 2019; Ippel, Kaptein, & Vermunt, 2016b). Chapter 3 raises the concern of the lack of research on MAB policies for dependent observations. In fact, dependencies are typically ignored and observations are treated as independent (also called *complete pooling*). The literature researching the more general contextual multi-armed bandit contains models for clustered observations from, for example, a single subject (also called *no pooling*), but potential dependencies between subjects are often ignored (Li, Chu, Langford, & Schapire, 2010). Chapter 3 tackles this issue by creating policies using a middle ground between no pooling and complete pooling approaches (also called *partial pooling*) – a method that is well known in the statistical literature (Efron & Morris, 1975; Gelman & Hill, 2006; Ippel et al., 2019; James & Stein, 1961). For this purpose, we develop streaming shrinkage factors (Ippel et al., 2019) for two types of policies, and a hierarchical Bayesian model (Gelman et al., 2013) for another policy. We empirically validate these novel treatment allocation procedures in a thorough simulation study and in an offline evaluation study using existing empirical data. Both studies show that incorporating partial pooling methods improves the performance of the multi-armed bandit policies. Using partial pooling in our example experiment would give us a great benefit for personalizing goals for (new) subjects; that is, start with a goal that turns out to work well on average for all subjects, but as more data comes in for the subject concerned, the partial pooling incorporates this information into (eventually) a personalized goal.

Before deploying a newly developed sequential allocation procedure in a real world experiment, its performance should be validated and evaluated according to the intricacies of the environment in which the experiment takes place. These intricacies could for example be the assumed distributions of the rewards or the distributions of the context. One option is to run multiple field evaluations, which is something that is technically easy to do with **StreamingBandit**, but in practice is often too expensive. Another option is to resort to simulation based methods, but these often lack external validity because of the (unrealistic) assumptions one needs to make. A better alternative is to evaluate allocation procedures using data collected in earlier experiments. One problem faced with using collected data is the omission of counterfactuals of the observed treatments (Li et al., 2011); that is, it is not known what would have happened if another treatment would have been allocated, simply because we are not able to allocate multiple treatments at once. As a consequence, if we would replay the whole data set row-by-row and ask a procedure to pick a treatment at a certain timepoint, it may well be that the current row does not contain any information regarding the selected treatment (e.g. treatment A is in row t of the data set, but the procedure selected treatment B). Over the last decade, various types of so-called offline evaluation methods have been developed to combat this problem (see e.g., Dudík, Erhan, Langford, Li, et al., 2014; Dudík, Langford, & Li, 2011; Li et al., 2011; Mary, Preux, & Nicol, 2014; Wang, Agarwal, & Dudík, 2017). One approach simply involves skipping an observation if the treatment allocated by the to-be evaluated policy is not equal to the treatment in the available data set (Li et al., 2011). For our example experiment, this method is insufficient, because these can be used only with a discrete set of treatments. Modeling the treatment “the number of steps” as a continuous range seems appealing, but this is where traditional offline evaluation methods fail. This is because the probability that the treatment suggested by a policy under evaluation exactly matches the treatment in the existing data set tends to zero for a continuous variable. As a result, no observations will be accepted for evaluation and the evaluation of the policy will fail. In Chapter 4, we extend the method by Li et al. (2011) to resolve this problem, and evaluate this new method. Furthermore, we compare our method for evaluating so-called *dynamic* treatment allocation procedures with a method developed by Kallus and Zhou (2018) for the offline evaluation of so-called *static* treatment allocation procedures – in which parameters are not updated during the experiment. The new method developed and evaluated in Chapter 4 makes it possible to use existing field data to evaluate and improve sequential allocation procedures for continuous treatments without making extra costs.

Chapter 5 provides a step-by-step demonstration of the integration of **StreamingBandit** into a front-end web application to implement a sequential allocation procedure in an experiment. As we have seen before, the rise of the internet allows researchers to conduct experiments (such as surveys) quicker, easier, and cheaper than in the past (for some examples, see also Kaptein, Van Emden, & Iannuzzi, 2017; Kaptein, van Emden, & Iannuzzi, 2016b). Online platforms such as **Qualtrics**, **SurveyMonkey**, and Amazon's Mechanical Turk (**MTurk**) offer services to implement and conduct those experiments (Amazon, 2012; Qualtrics, 2005; SurveyMonkey, 1999), while front-end web applications such as **Qualtrics** also allow some form of integration of treatment allocation procedures in their platform. However, the treatment allocation procedures are often very limited (i.e., only allowing for uniform random allocation). In this chapter, it is shown how **StreamingBandit** can be used to deploy experiments with sequential allocation procedures in a front-end web application – where **Qualtrics** is chosen as an example platform. Once **StreamingBandit** is integrated into the desired platform, it allows the researcher to easily adapt experiments to use different types of allocation procedures. As an extra illustration, an implementation of a recently completed experiment is discussed in detail. For our example experiment, and given the tools developed in this thesis, it is easy to use **Qualtrics** to deploy an experiment, where for the necessary communication needed to provide daily goals, **Qualtrics** could be replaced by a platform that allows for direct communication with a smartwatch (e.g., existing applications). Thus, Chapter 5 shows hands-on how the methods developed in this thesis can be used to improve and extend social science experiments by making use of sequential treatment allocation procedures.

In the epilogue in Chapter 6, we give a short overview of the work done in this thesis and discuss some of the limitations and potential trajectories for future research.

Chapter 2

StreamingBandit: Experimenting with Bandit Policies

Abstract

A large number of statistical decision problems in the social sciences and beyond can be framed as a (contextual) multi-armed bandit problem. However, it is notoriously hard to develop and evaluate policies that tackle these types of problems, and to use such policies in applied studies. To address this issue, this paper introduces **StreamingBandit**, a Python web application for developing and testing bandit policies in field studies. **StreamingBandit** can sequentially select treatments using (online) policies in real time. Once **StreamingBandit** is implemented in an applied context, different policies can be tested, altered, nested, and compared. **StreamingBandit** makes it easy to apply a multitude of bandit policies for sequential allocation in field experiments, and allows for the quick development and re-use of novel policies. In this article, we detail the implementation logic of **StreamingBandit** and provide several examples of its use.

Keywords: sequential decision-making, multi-armed bandit, data streams, sequential experimentation, Python

Based on Kruijswijk, J.M.A., Van Emden, R., Parvinen, P., Kaptein, M.C. (2020). StreamingBandit: Experimenting with Bandit Policies. *Journal of Statistical Software*.

2.1 Introduction

In the canonical multi-armed bandit (MAB) problem a gambler faces a number of slot machines, each with a potentially different payoff. It is the gambler's goal to make as much profit (or, in the case of gambling, as little loss) as possible by sequentially choosing which machine to play, learning from the observations as she goes along (Berry & Fristedt, 1985; Whittle, 1980). The gambler faces a trade-off between exploration and exploitation (Macready & Wolpert, 1998): on the one hand she wishes to play the machine that was successful in earlier attempts as often as possible (exploitation), but on the other hand she wishes to find the machine with the highest payoff through experimentation (exploration). The MAB problem, and its generalization, the contextual MAB (or CMAB) problem – in which before selecting a machine the gambler observes the state of the world that could be related to the optimal choice of machine at that point in time – provides a flexible formalization for studying sequential treatment-allocation procedures in the social sciences and beyond (Agrawal & Goyal, 2013b; Dudík, Hsu, et al., 2011; Li et al., 2011).

A multitude of policies addressing (contextual) decision problems have been conceived and evaluated (see, e.g., Berry & Fristedt, 1985; Chapelle & Li, 2011; Dudík, Hsu, et al., 2011). Indeed, the randomized controlled trial (RCT, or ϵ-first in the literature on sequential decision-making; Chapelle and Li 2011) is in itself a specific policy devised to address the exploration-exploitation trade-off in which an exploration phase, the trial itself, is followed by exploitation. Other policies range from simple heuristics such as "play the winner" (Lachin, Matts, & Wei, 1988; Villar, Bowden, & Wason, 2015) to asymptotically optimal policies such as upper confidence bound (UCB) methods (Audibert, Munos, & Szepesvári, 2009; Auer & Ortner, 2010; Garivier & Cappé, 2011), and Bayesian methods such as Thompson sampling (Agrawal & Goyal, 2012; Chapelle & Li, 2011; Thompson, 1933). It is difficult to assess which of these policies performs best in distinct applied problems, however, due to the omission of the counterfactuals in the (field) evaluations of a policy (Li et al., 2011): one does not know what the outcome would have been had another choice been made anywhere along the sequence of decisions. Hence the data resulting from an evaluation can often not be used to evaluate alternative policies. To evaluate a range of possible policies one has to resort to either simulation methods – which often lack external validity due to the large number of assumptions encoded in the simulation – or to recent offline evaluation methods (Agarwal et al., 2017; Li et al., 2011). Offline methods provide the opportunity to obtain unbiased estimates of the performance of different policies on historical data, but these approaches are only practically feasible when the number of choice alternatives is relatively low and/or the number of sequential choices is large. Furthermore, the assumptions that justify these methods – such as stationarity and a non-zero probability for each possible treatment at each interaction (Li et al., 2011) – are rarely fully justified in practice.

Despite these difficulties, effective (contextual) decision policies are potentially of great use in many areas. To unleash this potential, researchers need to be able quickly to implement and evaluate distinct bandit policies in the field. This can be achieved by allowing substantive researchers to easily test different sequential allocation schemes. If easy-to-use software were available for evaluating and disseminating novel policies, such policies – which are actively being developed (e.g., Bastani & Bayati, 2020; Eckles & Kaptein, 2014; Osband & Roy, 2015) – would be within reach of a broader research community. It is to this end that we developed **StreamingBandit**: an open-source RESTful web application that allows researchers to formalize their sequential-allocation procedure as a CMAB problem and, by virtue of this formalization, easily to experiment with different policies.

In the remainder of this section we first engage in a high-level discussion of the basic usage of **StreamingBandit**, discuss related approaches, and provide an overview of the application and its installation. In Section 2.2, we describe the application in more detail, and demonstrate the setup and evaluation of a single policy. Here we also discuss the use of **StreamingBandit** for offline policy evaluation and we offer a number of performance measures. In Section 2.3, we introduce a number of currently implemented "default" policies and discuss methods of combining multiple policies. We detail two practical applications of **StreamingBandit** in Section 2.4, and finally in Section 2.5 we briefly discuss future work directions.

2.1.1 Basic usage

The basic setting we consider is the following. Consider an experimenter who interacts with the environment. At each interaction t:

1. the experimenter observes a context x_t,
2. subsequently, the experimenter chooses an action a_t,
3. and finally a reward r_t is observed.

The main aim of the experimenter is to maximize the cumulative reward $\sum_{t=1}^{N} r_t$ where N denotes the total number of interactions. To do so, the experimenter applies a policy Π which is some function that takes the context x_t and the historical interactions, and returns an action. For convenience we denote all historical interactions using $\mathcal{D}_{(t-1)}$ and thus we have $\Pi(x_t, \mathcal{D}_{t-1}) \rightarrow a_t$.

This sequential decision-making scheme is encountered in many real-life situations:

- *Personalized healthcare:* A physician meets with patients sequentially. For each patient, she observes a number of background characteristics (gender, age, current condition) constituting the context. Subsequently, her action is to choose a treatment such that the reward – measured in terms of the general health of the patient – is maximized.
- *Online advertising:* In online advertising a firm selecting an ad observes the context consisting of a description of the current user visiting a specific webpage. The action is to choose an advertisement from of a set of possible advertisements (possibly dependent on the context), and the rewards constitute the clicks on the ad.
- *Product-recommendation systems:* The context denotes all that is known about the user at a certain point in time. The action is choosing one of a set of products, and the reward consists of the revenue generated at each interaction.
- *Social-science experiments:* Many social-science experiments constitute a special case of contextual decision-making: participants are recruited sequentially during the experiment. The context consists of all that is known about the participant, and sequentially the action is to assign a participant to a specific experimental condition (possibly dependent on the context in cases of stratified sampling, for example). Finally, the reward(s) consist of the outcome measures of the experiments.

The above list illustrates the generality of our approach: **StreamingBandit** can be used to allocate actions in all of the above applications.

To ensure the computational scalability of **StreamingBandit** we assume that, at the latest interaction $t = t'$, all the information necessary to choose an action can be summarized using a limited set of parameters denoted $\theta_{t'}$, the dimensionality of θ_t often being (much) smaller than that of $\mathcal{D}_{t-1}$. Given this assumption, we identify the following two steps of a policy:

1. *The decision step:* In the decision step, using $x_{t'}$ and $\theta_{t'}$, and often using some (statistical) model relating the actions, the context, and the reward, which is parametrized by $\theta_{t'}$, the next action $a_{t'}$ is selected. Making a request to **StreamingBandit**'s `getaction` REST endpoint returns a JSON object containing the selected action. Optionally, the probability $p_{t'}$ of selecting this action (the `propensity`) and/or an identifier for this specific request (the `advice_id`), both of which are explained in more detail below, are also returned.

2. *The summary step:* In each summary step $\theta_{t'}$ is updated using the new information $\{x_{t'}, a_{t'}, r_{t'}, p_{t'}\}$. Thus, $\theta_{t'+1} = g(\theta_{t'}, x_{t'}, a_{t'}, r_{t'}, p_{t'})$ where $g()$ is some update function. Effectively, all the prior data, $\mathcal{D}_{t-1}$ are summarized in $\theta_{t'}$. This choice means that the computations are bounded by the dimension of θ and the time required to update θ instead of growing as a function of t. Note that this effectively forces users to implement an online policy (Michalak et al., 2012) as the complete dataset $\mathcal{D}_{t-1}$ is not revisited at subsequent interactions. Making a request to **StreamingBandit**'s `setreward` endpoint containing a JSON object including either the `advice_id` or a complete description of $\{x_{t'}, a_{t'}, p_{t'}\}$, and the reward $r_{t'}$, allows one to update $\theta_{t'+1}$ and subsequently to influence the actions selected at $t' + 1$.

For the basic usage of **StreamingBandit** the experimenter – or rather an external server or mobile application – sequentially executes requests to the `getaction` and `setreward` endpoints, and allocates actions accordingly. Using this setup, **StreamingBandit** can be used to sequentially select advertisements on webpages, for example, allocate research subjects to different experimental conditions in an online experiment, or sequentially optimize the feedback provided to users off a mobile eHealth application. We provide a number of practical examples in Section 2.4.

2.1.2 Related approaches

Theoretically, contextual decision-making relates to a broad literature ranging from active learning (e.g., Beygelzimer, Hsu, Langford, & Zhang, 2010; Hanneke, 2014) to the general setting of reinforcement learning (Sutton & Barto, 2011; Szepesvári, 2010). The contextual MAB problem (Agarwal et al., 2014; Dudík, Hsu, et al., 2011; Li et al., 2010) we consider here is a specific instance of reinforcement learning: it is a problem that is well-studied both without contextual information (Berry & Fristedt, 1985) and in numerous generalizations, such as the continuous bandit (Mandelbaum, 1987) and bandits with dependencies (Pandey, Chakrabarti, & Agarwal, 2007). The current work also relates to recent discussions on offline policy evaluation (Dudík, Erhan, Langford, & Li, 2012; Dudík, Langford, & Li, 2011), although it is distinct from the multi-world testing service presented by Agarwal et al. (2017) in its focus on running (adaptive) policies online versus the online collection of data combined with the offline evaluation of policies. The field is too large to be properly reviewed in this paper, and we refer the reader to Schwartz, Bradlow, and Fader (2017) and the references therein for an accessible introduction and contemporary applications.

Here we narrow our discussion of related approaches to related software projects, which we split into the following four categories: i) software for A/B testing, ii) software for general (supervised) learning, iii) software for offline policy evaluation, and iv) software for (sequential) optimization. The first category relates to our current project in that A/B tests – or randomized experiments – are used in many fields to address (C)MAB problems: one devotes a (pre-set) number of interactions to random exploration, after which the best performing action is selected and further exploited. This approach has become standard in many web companies (Jiang, Shi, Shang, Geng, & Glass, 2016). A more advanced version, often referred to as "multi-variate testing" runs many A/B tests in parallel, possibly exploiting a factorial structure between the actions. Several commercial systems, such as Google **Analytics**, provide A/B testing abilities (Google, 2018), see also **Optimizely** (Optimizely, 2017), and **Mixpanel** (Mixpanel, 2017).

An effective policy depends heavily on the ability to predict the next reward given a context. Once available, a (large) dataset of contexts, actions, and rewards constitutes a supervised learning problem. Many general supervised learning solutions have been developed recently, such as **CNTK** (Seide & Agarwal, 2016), **GraphLab** (Collet, Sassolas, Lhuillier, Sirdey, & Carlier, 2016), **GeePS** (Cui, Zhang, Ganger, Gibbons, & Xing, 2016), **MLlib** (Meng et al., 2016), **TensorFlow** (Abadi et al., 2016), and **Minerva** (Reagen et al., 2016). Some of these, such as **Vowpal Rabbit** (Langford, Li, & Strehl, 2011) and **Jubatus** (Hido, Tokui, & Oda, 2013), explicitly include libraries implementing specific bandit policies, or evaluation methods for bandit policies on existing, offline, data sets. Specific software projects for offline policy evaluation, and hence the ability to evaluate policies on existing datasets, are also available (see, e.g., Komiyama, Honda, & Nakagawa, 2015; Nugent, 2015; Striatum Contributors, 2016). Others have provided language-specific code libraries implementing different policies, although most of these efforts seem to be a) geared towards computer scientists and experienced developers and b) not focused on field deployment (see Galbraith, 2016; Kaufmann, Cappé, & Garivier, 2012; Sola, 2015, and the references therein).

There are a number of platforms that allow for sequential optimization: Google **Analytics** (Google, 2018), for example, supports Thompson sampling (Agrawal & Goyal, 2012; Kaptein, 2014; Thompson, 1933), which is a method for sequentially allocating visitors to different actions dynamically based on the observed outcomes. However, contextual knowledge is not included. Yelp **MOE** (Yelp, 2014) is an open-source software package that implements optimization over a large parameter space via sequential A/B tests in which Bayesian optimization is used to compute parameters for the next best A/B test. Finally, the **Decision Service** (Agarwal et al., 2017) implements a number of functionalities im-

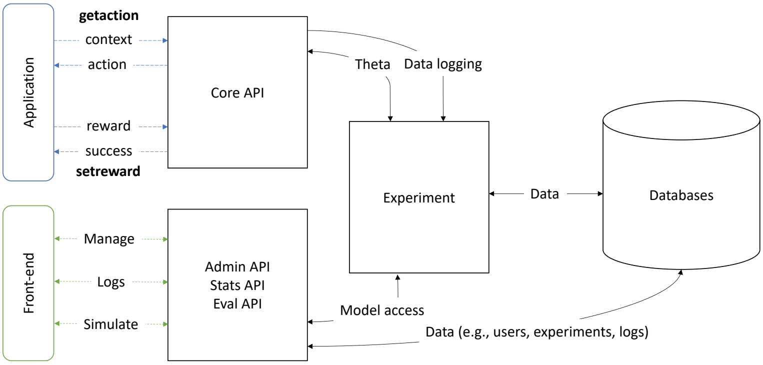


Figure 2.1: High-level architecture of **StreamingBandit**.

plemented by **StreamingBandit** using a similar formalization (the summary and decision steps). This software package focuses on continuously collecting data to update and deploy policies that are evaluated offline, whereas **StreamingBandit** focuses on evaluating (adaptive) policies online.

2.1.3 An overview of StreamingBandit API calls

StreamingBandit is a Python 3 application that runs a **Tornado** web server (Tornado Authors, 2016) and discloses a REST API that facilitates the implementation of the summary and decision steps described above. A user of **StreamingBandit** first creates an experiment and subsequently implements – or adopts based on the library of available policies – a policy using Python 3. A policy specification consists of a) some code implementing the decision step given $\theta_{t'}$ and $x_{t'}$, and b) some code implementing the summary step given the observed outcomes to update $\theta_{t'}$. Figure 2.1 presents an overview of the architecture of **StreamingBandit**. The application discloses a number of REST endpoints to facilitate the creation and editing of experiments and the extraction of data from running experiments. All endpoints apart from the `getaction` and `setreward` require the user to authenticate using a secure cookie. Logging in can be done by passing a JSON object to the `login` endpoint containing the parameters `username` and `password`; if the username and password are valid, a secure cookie is returned. New users can be created using the `user` call and posting the relevant information. For convenience, we provide a separate UI (a separate software project that can be found at https://github.com/Nth-iteration-labs/streamingbandit-ui) that allows easy point-and-click administration and management of experiments. Here we detail the primary endpoints and describe their functionality. We have already introduced the `getaction` and `setreward` calls, of which the full specification is:

GET `getaction`: the query-string parameters consist of the experiment identification number, `exp_id` (string), a `key` (string), and the `context` (JSON). The call executes the decision step of a policy associated with the `exp_id` and returns an `action` (JSON), which optionally contains the elements `advice_id` (string), and `propensity` (float). The `key` is used to authenticate the request.

GET `setreward`: the query-string parameters consist of the `exp_id`, the `key`, the `reward` (JSON) and either the `advice_id`, in which case the `context` and `action` are retrieved from the associated `getaction` call, or the `context` and `action` themselves. Subsequently, the summary step of the policy associated with the associated `exp_id` is executed and a JSON object containing the status is returned.

The primary REST endpoints at which to manage the experiments are:

GET `exp`: Returns a JSON object listing the `exp_id` and `name` of each experiment.

POST `exp`: Posting a JSON object containing the parameters `name`, `getcontext`, `getaction`, `getreward` and `setreward` creates a new experiment. The last four fields should contain executable restricted Python 3 code. To ensure some safety in the executed code we limit the functionality of these customer scripts to a subset of Python 3 code, using self-defined built-ins. This will disallow, for instance, the import of any other packages apart from the one we already make available. It also means that the user does not need to import any packages into the code because they are made available in the built-ins before any code is executed. The code in the `getaction` and `setreward` fields implements the decision and summary steps, respectively. The `exp` endpoint accepts a number of optional parameters, which we detail in Section 2.2.1. A valid POST request to the `exp` endpoint returns a JSON object containing the `exp_id` and the `key` of the newly created experiment.

The code in the `getcontext` and `getreward` fields is not strictly necessary; these two snippets of code provide for the opportunity to simulate sequential decisions. This is extremely useful for debugging and can be used in simulation studies of a policy. Passing the query-string parameter `n` (int, default=1) to rest endpoint `eval/<exp-id>/simulate` sequentially executes the `getcontext`, `getaction`, `getreward` and `setreward` code of the associated experiment n times.

PUT `exp/<exp_id>`: If the `exp_id` string in the url is a valid experiment for the current user, this call edits the existing experiment. The parameters are the same as those used for creating experiments.

GET `exp/<exp_id>`: Returns the name and `getaction` and `setreward` code for a specific experiment.

DELETE `exp/<exp_id>`: Deletes an experiment. When an experiment is deleted all the user-generated settings are removed, as well as the current θ. However, logged data associated with the experiment is maintained.

GET `exp/<exp_id>/resetexperiment`: Resets the experiment: the current state of θ is deleted, but all the other information is retained and the policy can still be executed.

Next to these administrative calls, the application provides a number of calls to monitor running experiments and retrieve logged data.

GET `stats/<exp_id>/currenttheta`: Returns the current θ for the experiment as a JSON object.

GET `stats/<exp_id>/summary`: Returns an overview of the number of requests to the `getaction` and `setreward` endpoints.

GET `stats/<exp_id>/rewardlog`: Returns the logged `setreward` events (including the `context`, `action`, and `reward` objects) for the current experiment. It can be used for offline policy evaluation (see, e.g., Agarwal et al., 2017; Li et al., 2011). The `limit` (int) query-string parameter limits the dump to the last k events.

GET `stats/<exp_id>/actionlog`: Returns all the `getaction` events for the current experiment. Again, the `limit` parameters limit the dump to the last k events.

GET `stats/<exp_id>/log`: Returns a JSON file of all data that was explicitly logged by the user using `self.log()` in the policy specification of an experiment.

Requests made to non-existing REST endpoints result in a 404 status error, whereas erroneous calls to existing end-points return a JSON object containing a key `error` with an informative error message.

2.1.3.1 Implemented policies: "defaults"

StreamingBandit comes with a number of implemented policies to tackle standard (contextual) decision problems. A JSON object containing a list of defaults can be retrieved using the endpoint `default`, and calling `default/<default_id>` gives the code for a specific default. We have implemented the following policies, amongst others:

- ϵ-first: Implements the standard randomized clinical trial approach to the (C)MAB problem: the first $t < n$ interactions, where n is set by the user, are allocated to actions randomly, after which the action with the highest average reward is selected for the remaining interactions.

- ϵ-greedy: Implements a greedy policy in which a proportion p of interactions is randomly allocated to the available actions, whereas a proportion of $(1-p)$ interactions is allocated to the action with the highest average reward at that point in time.

- Thompson sampling for the k-armed Bernoulli bandit: Thompson sampling provides a Bayesian solution to the MAB problem (Agrawal & Goyal, 2012; Thompson, 1933). We implement Thompson sampling for the Bernoulli bandit (e.g., $r \in \{0, 1\}$). Thompson sampling allocates actions proportional to one's current belief – as quantified using a posterior distribution – that an arm is optimal (Kaptein, 2014).

- Lock-in Feedback: Lock-in Feedback (LiF) is an allocation scheme for dealing with continuous actions ($a \in \mathbb{R}$) in which small systematic oscillations in the action choice over time are used to derive the gradient of the reward function and take a step toward the (local) maximum of that function (see Kaptein, van Emden, & Iannuzzi, 2016a; Kaptein, van Emden, & Iannuzzi, 2016b, for details).

- Bootstrap Thompson sampling: Bootstrap Thompson sampling provides a computationally appealing alternative to Thompson sampling in cases in which it is hard to sample directly from the posterior distribution of a model online (see Eckles & Kaptein, 2014). In essence, the posterior distribution is approximated using an online bootstrap distribution (Owen & Eckles, 2012).

We provide examples of the use of these policies in Section 2.3. **StreamingBandit** is easily extended and new defaults can be added by adding code to the `/defaults` folder of the application in a folder with an informative name that contains the following four files:

1. `get_context.py`: A Python script that generates a JSON object encoding a context.

2. `get_action.py`: A script that takes a JSON object encoding the context, and returns a JSON object containing the action.

3. `get_reward.py`: A script that generates a reward using a context and action JSON.

4. `set_reward.py`: A script that takes a context, action, and reward JSON and handles the logic of updating θ.

Restarting the web application after adding these files will automatically include the novel policy in the list of defaults. We welcome submissions of new default policies and other implementations. See Section 2.1.5 for more details.

2.1.3.2 StreamingBandit libraries

StreamingBandit was created to quickly create and test alternative policies in the field. This can be done by altering the `getaction` and `setreward` codes associated with an experiment. However, given that a number of operations are often encountered in the online processing of incoming data, **StreamingBandit** also provides a number of `Python` modules:

- `base`: This module provides functionalities for online (row-by-row) updates of, e.g., counts, means, variances, proportions, and covariances.
- `lm`: Implements an online version of a linear regression model.
- `bts`: Takes a model (e.g., `lm`) and a row of data and produces (or updates) an online bootstrap distribution of the parameters.
- `lif`: Implements the Lock-in Feedback policy, as described in Kaptein and Ianuzzi (2016).
- `thompson`: Implements Thompson sampling for the k-armed Bernoulli bandit, amongst others.
- `thompson_bayes_linear`: Implements model-based Thompson sampling using a Bayesian linear regression model.

New modules can be added to the application by adding a script to `/libs`. For detailed documentation of the individual modules we refer the reader to http://nth-iteration-labs.github.io/streamingbandit/libs.html.

2.1.4 Installation, deployment, and documentation

The **StreamingBandit** source code is available from https://github.com/Nth-iteration-labs/streamingbandit/ and the documentation can be accessed on http://nth-iteration-labs.github.io/streamingbandit/. There are several ways in which **StreamingBandit** can be used:

1. At http://sb.nth-iteration.com we provide a running instance of **StreamingBandit**. You apply for a user account by sending an email to the corresponding author of this paper, and use our hosted webserver for (small-to-medium-sized) projects.

2. The easiest way to get going independently is probably to use our **Docker** container (Merkel, 2014). The following commands assume that you have **docker** and **docker-compose** installed, and that you are inside a folder in which you wish to put the source code of **StreamingBandit**.[1] If so, starting **StreamingBandit** requires, first, pulling the repository to your local system and going inside the folder:

   ```
   $ git clone \
   > http://github.com/Nth-iteration-labs/streamingbandit.git
   $ cd streamingbandit
   ```

   Next, once you are inside the folder with all the source code, we can launch **StreamingBandit** by running:

   ```
   $ docker-compose up -d
   $ docker exec -t streamingbandit_web_1 python3 \
   > ../insert_admin.py -p test
   ```

   The first command makes sure that all necessary containers, including the databases, are running. The second command creates a user account admin with the password "test". To gracefully stop and start the container after running the first command, run the following command:

   ```
   $ docker-compose stop
   $ docker-compose start
   ```

   Starting the service will make **StreamingBandit** available at http://localhost:8080 or the **Docker**-set IP address.

   Note that the above commands only start the back-end REST service. The following commands are also needed to launch our front-end:

   ```
   $ docker-compose -f docker-compose.yml \
   > -f docker-compose.front-end.yml up -d
   $ docker exec -t streamingbandit_web_1 python3 \
   > ../insert_admin.py -p test
   ```

   The start and stop commands now change slightly as well:

[1] For more information on how to get started with **Docker**, see https://docs.docker.com/get-started/.

```
$ docker-compose -f docker-compose.yml \
> -f docker-compose.front-end.yml stop
$ docker-compose -f docker-compose.yml \
> -f docker-compose.front-end.yml start
```

which starts and stops both the front-end and the back-end at the same time. The front-end can be reached at http://localhost or the **Docker**-set IP address.

The front-end source-code can be found in a separate repository at https://github.com/Nth-iteration-labs/streamingbandit-ui, but for this use-case it is not necessary to download the repository to your local system because we have uploaded a **Docker** image to the internet and **Docker** will download that image automatically via the `docker-compose` command.

3. For larger-scale projects we recommend installing from source and perhaps using a load-balancer. For details, please consult the documentation at http://nth-iteration-labs.github.io/streamingbandit/.

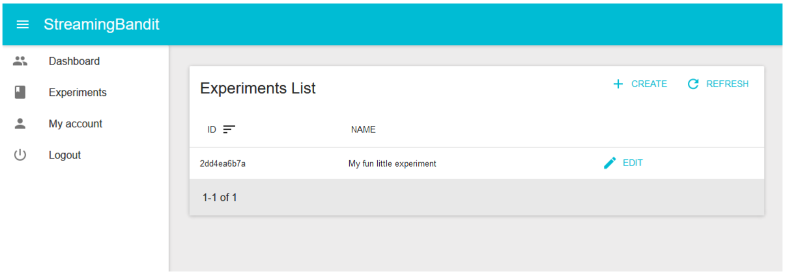
2.1.5 Further development

The above sections give the essential details of **StreamingBandit**. We gladly accept any contributions towards making **StreamingBandit** better and more useful. The guidelines for contributing to the development of **StreamingBandit** can be found in the documentation.

2.2 Getting started

In the remainder of this article we assume that the reader is running the default **Docker** container installation of **StreamingBandit**, and is using the management front-end for the administration of experiments. In introducing the details of setting up a policy we describe the setup and usage of a simple – but very frequently used – policy: ϵ-first. When this policy is executed a sample of size n interactions is uniformly randomly allocated to a control ($a =$ control) or treatment ($a =$ treatment) action (or condition), after which the treatment is adopted if it is more effective than the control condition. With slight abuse of the notation this can be denoted:

$$\Pi_{\texttt{ϵ-first}}(\mathcal{D}, n) = \begin{cases} a_t \sim \texttt{random}(\text{control}, \text{treatment}) & \text{if } t \leq, n \\ a_t = \texttt{max}(\bar{r}_{\text{control}}, \bar{r}_{\text{treatment}}) & \text{otherwise}, \end{cases} \tag{2.1}$$



Experiments List

ID	NAME
2dd4ea6b7a	My fun little experiment

1-1 of 1

Figure 2.2: Screenshot of the default front-end for **StreamingBandit**.

where $\bar{r}_{\text{control}}$ denotes the sample average of outcomes observed in the control condition when $t \leq n$, and the last line denotes selection of the action with the highest empirical average reward when $t > n$. The management front-end – of which Figure 2.2 shows a screenshot – makes it easy to create a new experiment or to use one of the defaults as a starting point for creating one's own policies. We present the front-end in more detail in Appendix 2.A. Once the experiment has been created it receives an `exp_id` and a key `key`. This enables the REST endpoints

```
http://HOST/getaction/<exp_id>?key=<key>&context={}
```

and

```
http://HOST/setreward/<exp_id>?key=<key>&context={}&reward={}&actio⌋
↪ n={}.
```

The actual functionality is provided by the `getaction` and `setreward` code specified when the experiment is created, whereas the `getcontext` and `getreward` codes are useful for simulations and testing. Below we detail each of these in turn for the version of ϵ-first implemented in the defaults. Before that we should note that we will denote a few variables and functions using `self` inside the code. These variables and functions are denoted with `self` because they are part of the experiment class in which the custom code runs. For the most part, we will only use a reference to `self` with the following variables and functions:

- `self.context`
- `self.action`
- `self.reward`
- `self.get_theta()`
- `self.set_theta()`

The code for a simple ϵ-first implementation is as follows:

- `getcontext`: The canonical ϵ-first strategy does not consider a context. Hence, we leave this blank.

- `getaction`: The implementation of the decision step of ϵ-first is:

```
n = 100
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if mean_list.count() >= n:
    self.action["treatment"] = mean_list.max()
    self.action["propensity"] = 1
else:
    self.action["treatment"] = mean_list.random()
    self.action["propensity"] = 0.5
```

  This code uses a number of libraries implemented in **StreamingBandit**: below we detail each line in turn. First, the sample size of the experiment, n in Equation 2.1, is set. The next line of code generates a list of `base.Mean` objects. This object provides the functionality to compute streaming updates of sample averages, and the list contains one such average for each of the possible treatments specified by name, using `["control", "treatment"]`. The `self.get_theta()` call is used to retrieve $\theta_{t'}$, which in this case thus contains two `base.Mean` objects named "control" and "treatment". A count, n, and mean reward, $\bar{r}$, are contained within each `base.Mean` object.

  The resulting `mean_list` object thus, in this case, contains two `base.Mean` objects, each of which contains a mean value and a count that can be updated and manipulated. In the next lines the total count of the number of observations over all mean elements in the list is retrieved. If this is larger than n, the treatment with the highest average value is returned, otherwise a random element of the list is returned. The returned JSON object when making a call to http://HOST/<exp_id>/getaction?key=<key> and filling in the correct `exp_id` and `key` appears as follows:

```
{"action":
     {"treatment": "control",
     "propensity": 0.5},
"context": {}}
```

  where the value of `treatment` changes randomly as long as $n \leq t$.

- `getreward`: Rewards can be simulated by using a few lines of Python 3 code

```
if self.action["treatment"] == "control":
    self.reward["value"] = np.random.normal(4, 1)
else:
    self.reward["value"] = np.random.normal(6, 2)
```

in which the rewards for the selected action in the decision step are drawn from a normal distribution ($r_{\text{control}} \sim \mathcal{N}(4, 1)$, $r_{\text{treatment}} \sim \mathcal{N}(6, 2)$).

- `setreward`: When a reward has been generated, the summary step for the ϵ-first policy is implemented as:

```
n = 100
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if mean_list.count() < n:
    mean = base.Mean(self.get_theta(key = "treatment",
        value = self.action["treatment"]))
    mean.update(self.reward["value"])
    self.set_theta(mean, key = "treatment",
        value = self.action["treatment"])
```

which again uses the **libs.base** library. After this the action is retrieved and the associated mean object is updated using `mean.update` as long as the exploration phase is ongoing. The last line stores $\theta_{t'+1}$ such that it can be retrieved again for future decision-making. In this implementation, after the experiment when $n > t$, θ is no longer updated. Note that a slightly more elaborate version of this example that facilitates propensity scores (see Section 2.2.1) can be found in the defaults (see Section 2.1.3).

As stated above, the `getcontext` and `getreward` codes are not strictly necessary to use the implemented policy in field studies; these two snippets of code merely provide the opportunity to simulate an experiment, a feature that is extremely useful for debugging. In actual evaluations of a policy the data resulting from these calls would be sent by the outside world (e.g., via a website or mobile application). However, to demonstrate the utility of the `getcontext` and `getreward` codes, note that a request to the endpoint `/eval/<exp_id>/simulate` with parameters `N=150`, `seed=1271246`, and `verbose=False` yields the following JSON response:

```
{
    "theta": {
        "treatment:control": {
            "n": "52",
            "m": "4.0259030511640885"
        },
        "treatment:treatment": {
            "n": "48",
            "m": "5.829777419810004"
        }
    },
    "simulate": "success",
    "experiment": "121e3e0aeb"
}
```

which shows the number of times the `treatment` and `control` conditions were selected (`n`) and their respective mean reward (`m`). Although we simulated 150 interactions, the total number of interactions stored in θ is $48 + 52 = 100$ because in the implementation above we stop updating θ when $t > n$.

2.2.1 Additional features

We described the setup and simulation of a simple bandit experiment in the previous section. The description skipped over a number of useful features of **StreamingBandit**, which we address below.

2.2.1.1 Offline analysis of bandit policies

When we first introduced the `getaction` endpoint we mentioned the optional return field `propensity`. In a number of default policies, the return object contains this propensity p_t, which is the probability of selecting the action at interaction t. By way of an illustration, for ϵ-first, as detailed above, the computation of p_t is as follows:

$$p_t = \begin{cases} 0.5 & \text{if } t \leq n \\ 1 & \text{otherwise} \end{cases}$$

Whenever it is possible to compute these propensities – which is sometimes difficult, such as when $a \in \mathbb{R}$ – the default policies include p_t. This serves two purposes:

1. When addressing contextual sequential decision problems, and when the probability of selecting an action depends on the context, the propensity p_t can be used for inverse propensity matching or weighting (Austin, 2011) to improve the estimate of the causal effect of the action by accounting for the contextual covariates (see, e.g., Imbens & Rubin, 2015; Pearl, 2009).

2. When p_t is included, the logged data of an experiment can be used for the offline evaluation of alternative decision policies. This can be attained by using inverse propensity scoring (ips). Suppose we are evaluating a policy Π using a logged dataset containing N events. The ips estimate of average reward of the policy can be obtained by computing

   $$\texttt{ips}(\Pi) = \frac{1}{N} \sum_{t=1}^{N} \mathbb{1}\{\Pi(x_t) = a_t\} r_t / p_t$$

   where the indicator $\mathbb{1}$ is 1 when the action of Π matches the action in the logs. Agarwal et al. (2017) provide a more extensive discussion of the benefits of using offline methods to evaluate alternative policies.

2.2.1.2 Advice ID, delayed rewards, and logging

When we described the [POST] `exp` endpoint we omitted a number of optional parameters that can be supplied in the JSON object. First of all, we skipped discussion of the `advice_id` parameter. This Boolean indicates whether or not the `getaction` call should return an `advice_id`. When set to `True` the `advice_id` parameter enforces a direct link between the `getaction` and `setreward` endpoints. In the example discussed above we were implicitly assuming that the application consuming the REST API would handle the logic that ensures that by the time the `setreward` endpoint is called, the `context`, `action` (including the `propensity`), and `reward` are properly supplied. However, this could be challenging for some consuming applications. In such cases, setting `advice_id = True` would require the consuming application to merely specify the `advice_id` when making a request to the `setreward` endpoint; **StreamingBandit** will merge the actions and context that were provided earlier in the associated `getaction` call with the rewards supplied in the `setreward` call.

When setting `advice_id = True`, one can also specify a) how long the `advice_id` will be retained (in hours). This is useful in some specific applications. In an online advertising experiment, for example, when a click on an advertisement is not registered within 12 hours it is extremely unlikely that this will happen in the future; it is more likely that the appropriate call to the `setreward` with $r_t = 0$ failed to register. Setting `delta_hours=12` and `default_reward = {"reward":"0"}` ensures that after twelve hours the `setreward` call associated with the `advice_id` is automatically executed with a reward of zero. It should also be noted that although all the examples provided in this paper sequentially execute the `getaction` and `setreward` calls, this is not at all a necessity. However, any bias in a (learning) model that might originate from, e.g., a delay in the arriving data in the `setreward` calls should be explicitly handled by the user.

Finally, we have not yet discussed the `hourly_theta` Boolean: if this is set to `True` when creating the experiment, the state of θ will be logged every hour. Calling `stats/<exp_id>/hourly_theta` with parameter `limit` returns the last k of these snapshots of θ, which could be useful for monitoring the progress of an experiment over time.

2.2.1.3 The nesting of policies

In addition to the libraries described earlier, and the `self.get_theta()` and `self.set_theta()` methods for storing and retrieving data, there are a number of methods available to the user from the code supplied in the `getaction` and `setreward` fields. The most interesting of these is the ability to instantiate other experiments within a running experiment. By way of illustration, the code

```
experiment = Experiment(exp_id = <exp_id>)
self.action = experiment.run_action_code(context = self.context)
```

can be used to run the `getaction` code of the experiment with `exp_id=<exp_id>` from another experiment. Similarly, `experiment.run_reward_code()` would execute the `setreward` code for another experiment. This allows the user to nest different experiments, and hence to essentially use a sequential decision policy Π^* to decide from among a range of policies that are being executed $\Pi_{1,\ldots,k}$. We provide a working example of this policy nesting in the Section 2.3.6.

2.2.2 Performance

To examine the performance of our RESTful API we set up an Ubuntu 16.04 x64 quad-core virtual server with 16GB of RAM running the **StreamingBandit** server, and additionally installed the **wrk2** load generator on a smaller (single core, 1GB RAM) Ubuntu 16.04 x64 machine connected to the same subnet within the same datacenter. We chose **wrk2** (Tene, 2015) as our load generator, as it is a HTTP benchmarking tool that is capable of generating significant load when run on a single CPU, and can easily be extended to test different RESTful HTTP methods through the use of `Lua` (Ierusalimschy, 2016) scripts.

To ensure that our load tests would not be hampered by OS related limitations we optimized sysctl.conf on both machines, turning off disk swapping, upping the number of connections per port, and optimizing port reuse. We also tested our client-server throughput with **iPerf3** (iPerf Authors, 2016). These tests indicated a throughput of 736 Mbits/s – more than enough bandwidth to safeguard against system-level I/O bottlenecks interfering with our API-level tests.

On completion of our test-bed we proceeded to run several **wrk2** load tests, focusing on industry-standard API performance measures (De, 2017). The results for a single **wrk2** thread running 100 concurrent AB test `getaction` calls at a time with a throughput limit of 1000 requests per second were the following:

- Average, max and standard deviation of latency: 21.09ms, 90.56ms, 13.22ms
- Throughput, in requests per second: 100 (equal to max set **wrk2** throughput)
- Top total CPU utilization: 69% (Of which: Python 3 65% of one of four available CPU's)
- Top Heap memory utilization: 3%

When we compared these numbers against some representative Python web framework benchmarks (Klenov, 2015) we found that **StreamingBandit** could hold its own. Still, to obtain a more objective measure of how "empty" versus "AB test" **StreamingBandit** `getaction` calls measure up to basic, vanilla **Tornado** requests, we compared these as well. The results, as illustrated in Figure 2.3, demonstrate that **StreamingBandit** adds little overhead to basic **Tornado** processing, and scales well up to 250 to 300 requests per second when running on a single virtual CPU core. The relatively minor increment in throughput and latency between the "empty" and the "AB test" experiments further indicates that **StreamingBandit** offers sufficient capacity to implement more complex experiments.

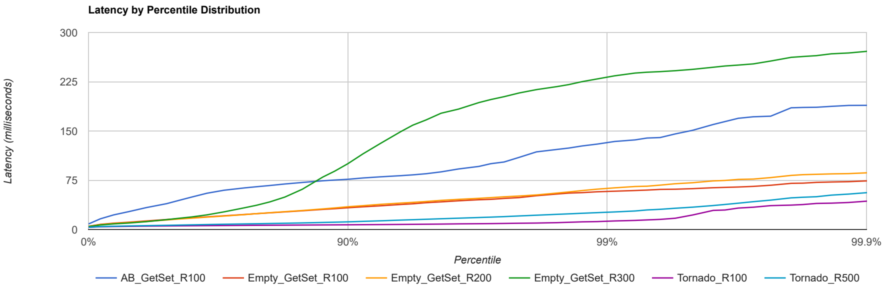


Figure 2.3: Latencies of basic **Tornado** calls when taxed by **wrk2** at a maximum throughput of 100 (Tornado_R100) versus 500 (Tornado_R500) calls per second (cps), as compared to **StreamingBandit** "AB test" (AB_GetSet_R100, throughput limited at 100 cps) and empty `getaction/setreward` calls (with Empty_GetSet_100, Empty_GetSet_200, Empty_GetSet_300 at respectively 100, 200 and 300 cps).

2.3 Examples of the implemented policies

In the following we work out a number of different (C)MAB policies. First, we present a simple implementation of ϵ-greedy (Sutton & Barto, 2011), then we introduce Thompson sampling for the canonical k armed Bernoulli bandit (Thompson, 1933), and for optimal design in between-subject experiments (Kaptein, 2014). We proceed by demonstrating two possible policies to deal with the continuum-bandit problem (problems in which $a \in \mathbb{R}$): Bootstrap Thompson sampling for a CMAB problem using a simple linear model (Eckles & Kaptein, 2014) and Lock-in Feedback (LiF) (Kaptein & Ianuzzi, 2016). We further demonstrate how **StreamingBandit** can be used to nest multiple policies, and show how **StreamingBandit** can be used to evaluate multiple policies in parallel using the offline evaluation method proposed by Li et al. (2011). This latter approach is, to the best of our knowledge, novel. All of the implementations discussed in this section can be found in the defaults (see Section 2.1.3).

2.3.1 ϵ-greedy

One frequently used policy is called ϵ-greedy (Sutton & Barto, 2011). It is implemented in a simple problem consisting of a control and a treatment arm, as we considered when we introduced ϵ-first, by playing the arms uniformly randomly with some probability ϵ, and selecting the hitherto best-performing arm with probability $1 - \epsilon$. The same `getcontext` and `setreward` codes as in our ϵ-first example above are used to implement ϵ-greedy, as follows:

- `getaction`:

```
e = .1
mean_list = base.List(self.get_theta(key = "treatment"),
    base.Mean, ["control", "treatment"])
if np.random.binomial(1,e) == 1:
    self.action["treatment"] = mean_list.random()
    self.action["propensity"] = 0.1*0.5
else:
    self.action["treatment"] = mean_list.max()
    self.action["propensity"] = (1-e)
```

Where, contrary to our ϵ-first example, we explicitly include the computation of the propensity p_t.

- `setreward`: The summary step for the ϵ-first can be implemented as:

```
mean = base.Mean(self.get_theta(key = "treatment",
    value = self.action["treatment"]))
mean.update(self.reward["value"])
self.set_theta(mean, key = "treatment",
    value = self.action["treatment"])
```

which is the same as for ϵ-greedy except for the fact that the respective means are updated at each interaction t instead of $n < t$.

Running a simulation with $n = 1000$ and `seed` $= 1271246$ gives:

```
{
    "theta": {
        "treatment:control": {
            "n": "70",
            "m": "4.011771491758239"
        },
        "treatment:treatment": {
            "n": "930",
            "m": "5.9609857188253015"
        }
    },
    "simulate": "success",
    "experiment": "3ea45886b5"
}
```

in which it is clear that the `treatment` arm is preferred.

2.3.2 Thompson sampling for the K-armed Bernoulli bandit

As our second example we provide the code to implement Thompson sampling for the classical Bernoulli bandit problem where the rewards are either 0 or 1, and for each arm $k = 1, \ldots, K$ the probability of success (reward = 1) is μ_k (Kaufmann, Korda, & Munos, 2012). Thompson sampling is a Bayesian policy in which one selects an action with a probability that is proportional to one's posterior belief that the action is optimal (see Kaufmann, Korda, & Munos, 2012, for details). In the Bernoulli reward case the Beta(α, β) distribution provides a convenient a priori choice in that after observing a Bernoulli trial the posterior distribution is simply Beta$(\alpha+1, \beta)$ in the case of success, and Beta$(\alpha, \beta+1)$ in the case of failure. Using S_k and F_k to denote the number of failures and successes for arm k, both of which are 0 at the start, Thompson sampling proceeds as follows; at each interaction t,

1. for each arm $k = 1, \ldots, K$, sample $d_k(t)$ from Beta$(S_k + 1, F_k + 1)$,
2. select arm $k(t) = \arg\max_k d_k(t)$,
3. and if $r_t = 1$ then $S_k = S_k + 1$ or when $r_t = 0$ then $F_k = F_k + 1$.

Thompson sampling for the 4-arm Bernoulli bandit problem can be implemented as follows:

- `getcontext`: The Bernoulli bandit does not consider a context; we leave this field blank.
- `getaction`: The decision step, using the **libs.thompson** library, can be implemented using:

```
propl = thmp.BBThompsonList(self.get_theta(key = "treatment"),
    ["1","2","3","4"])
self.action["treatment"] = propl.thompson()
self.action["propensity"] =
↪  propl.propensity(self.action["treatment"])
```

  where the four arms are indexed using the numbers $1 - 4$.

- `getreward`: Bernoulli rewards can be simulated using:

```
self.reward["value"] = np.random.binomial(1,
    (0.2*int(self.action["treatment"])))
```

which produces Bernoulli rewards with a probability of $0.2, 0.4, 0.6, 0.8$ for the four arms respectively.

- `setreward`: Finally, the updates of the posterior distributions are implemented using

```
prop = base.Proportion(self.get_theta(key = "treatment",
    value = self.action["treatment"]))
prop.update(self.reward["value"])
self.set_theta(prop, key = "treatment",
    value = self.action["treatment"])
```

Running a simulation with $n = 1000$ and `seed` $= 1271246$ gives:

```
{
    "theta": {
        "treatment:1": {
            "p": "0.14285714285714288",
            "n": "7"
        },
        "treatment:4": {
            "p": "0.8025404157043878",
            "n": "866"
        },
        "treatment:2": {
            "p": "0.20000000000000004",
            "n": "10"
        },
        "treatment:3": {
            "p": "0.5982905982905986",
            "n": "117"
        }
    },
    "experiment": "1de9753f51",
    "simulate": "success"
}
```

Which demonstrates that arm 4 is clearly, and correctly, preferred.

2.3.3 Thompson sampling for optimal design

Another example that could have practical relevance in social-science experiments is presented in Kaptein (2014): When running an experiment comparing two groups that receive different treatments, assuming unequal variances in the observed continuous outcomes, it is beneficial to allocate a larger number of subjects to the treatment with the highest variance to increase the precision in the obtained effect-size estimate. The Thompson sampling policy to implement this sequential allocation is to compute – using a normal-inverse χ^2 model – the posterior variances σ_1^2 and σ_2^2 of the two treatments in the summary step. Next, in the decision step, a draw d from each of the two posterior distributions σ_1^2 and σ_2^2 is obtained and the treatment is selected for which $\frac{d}{n}$, where n denotes the number of subjects allocated to the respective treatment, is highest. This choice leads to the largest reduction in the estimated standard error of the mean difference between the two groups. We refer the interested reader to Kaptein (2014) for details. This sequential allocation scheme can be implemented In **StreamingBandit** using:

- `getcontext`: Left blank as no context is considered
- `getaction`: In the summary step, we retrieve a list of two variance objects, one for each treatment. Variance objects, and the ability to update these online, are included in **base** library. Next, we implement Thompson sampling on the level of the posterior variances of the outcomes; this is included in the **libs.thompson** library:

```
varList = thmp.ThompsonVarList(self.get_theta(key =
↪   "treatment"),
    ["control","treatment"])
self.action["treatment"] = varList.experimentThompson()
```

- `getreward`: To simulate outcomes with unequal variances we can use:

```
if self.action["treatment"] == "control":
    self.reward["value"] = np.random.normal(0, 1)
else:
    self.reward["value"] = np.random.normal(1, 5)
```

- `setreward`: And finally, we update the respective posterior variance when new observations arrive:

```
var = base.Variance(self.get_theta(key =
↪  self.action["treatment"]))
var.update(self.reward["value"])
self.set_theta(var, key = "treatment", value =
↪  self.action["treatment"])
```

Running a simulation with $n = 100$ and `seed` $= 43123$ gives:

```
{
    "theta": {
        "treatment:treatment": {
            "s": "1453.3754330265062",
            "n": "77",
            "x_bar": "0.777831868342291",
            "v": "19.123360960875083"
        },
        "treatment:control": {
            "s": "32.31094303640007",
            "n": "23",
            "x_bar": "0.032257238191552844",
            "v": "1.4686792289272759"
        }
    },
    "experiment": "84b4d7eda",
    "simulate": "success"
}
```

This result highlights two things: First, it is clear that the treatment condition with the highest variance is indeed selected more often. This is the expected behavior to ensure that the precision of the estimate is increased. Second, the result demonstrates the internals of the `base.Variance` object: to compute a variance in a data stream we maintain a count (`n`), a mean (`m`), and the numbers `s` and `v`; of these `v` is the current sample variance, whereas `s` is an auxiliary variable used to implement Welford's method for computing a variance online (Welford, 1962).

2.3.4 Bootstrap Thompson sampling

Bootstrap Thompson sampling (BTS) is a recent approach devised to address CMAB problems (see, e.g., Eckles & Kaptein, 2014; Osband & Roy, 2015). The basic idea behind BTS is that instead of using a draw from the posterior distribution of the parameters of interest to decide on the next allocation, as is the case in previous Thompson sampling examples, one can maintain, online, a number of bootstrapped estimates of the parameters. These bootstrapped estimates can then be used to balance exploration and exploitation by randomly selecting one of the bootstrap replicates (see Eckles & Kaptein, 2014, for details).

StreamingBandit implements this sequential allocation scheme quite generally using the double-or-nothing bootstrap (Owen & Eckles, 2012). The appeal of BTS compared to traditional Thompson sampling is that a) it can be fully carried out online as long as the point estimates of interest can be obtained online, and b) it can be used in many situations in which obtaining draws from the true posterior density of interest is computationally difficult. Here we provide a simple example of the implementation of BTS using a linear model to relate the actions, the contexts, and the rewards.

For ease of exposition, let us consider a practical example. Suppose we are concerned with choosing a `price` (the action) of a product sold online such that the `revenue` is maximized (the reward). Let us further assume that we believe the relation between these two quantities is quadratic, and that we think the optimal sales price differs between `new` customers and `returning` customers. The following code implements this scenario such that it can be simulated:

- `getcontext`: The get context code simulates the visit of either a new or a returning visitor.

```
self.context["customer"] = random.choice(["new", "returning"])
```

- `getaction`: Next, the get action code, which is slightly more involved, uses the **lm** library to instantiate $m = 100$ linear models of the form

$$\begin{aligned}\texttt{revenue} \sim \beta_0 + \beta_1\texttt{price} + \beta_2\texttt{price}^2 + \beta_3\texttt{new} +& \\ \beta_4\texttt{price} \times \texttt{new} + \beta_5\texttt{price}^2 \times \texttt{new}.&\end{aligned} \tag{2.2}$$

  Here, the starting values of the model β's are initially set to zero. The `BTS` object maintains $m = 100$ of these models, whereas the remaining code samples one of these $m = 100$ models and computes the `price` that maximizes the expected revenue given the current customer and the current state of the parameters. We add comments to the code to improve readability:

```
# Instantiate BTS with m=100 samples:
BTS = bts.BTS(self.get_theta(), lm.LM, m = 100, default_params
↪   = {'b': np.zeros(6).tolist(), 'A' :
↪   np.identity(6).tolist(), 'n' : 0})

# Return one of the m samples:
model = lm.LM(default = BTS.sample())

# Retrieve its coefficients:
betas = model.get_coefs()

# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0

# Maximize the function
if betas[2] != 0 or betas[5] != 0:
    x = ( (-(betas[1] + betas[4] * customer)) /
        (2*(betas[2] + betas[5] * customer)) )
    x = np.asscalar(x)
    if x < 5 or x > 20:
        x = np.random.uniform(5,20)
else:
    x = np.random.uniform(5,20)

# Return the price
self.action["price"] = x
```

Note that we restrict the prices to be between 5 and 20, such that if BTS needs some more exploration, it will not go towards extreme values, which may happen if a linear model is selected that has no parabola – in a field experiment you might want to have your prices restricted to certain ranges as well.

- `getreward`: In the get reward code we use a logistic function to simulate the probabilities of accepting or rejecting the product at the offered price for different customer types.

```
# Get parameters
# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0
price = self.action["price"]

# Create logistic function
logistic = lambda x: 1 / (1 + numpy.exp(-x))

# Compute purchase yes / no
buy = numpy.random.binomial(1,
    logistic(-0.1 * (price - (10+4*customer))**2))

# Compute the reward
self.reward["revenue"] = buy * price
```

Here it is clear that new customers are more inclined than returning customers to buy for higher prices, the revenue-maximizing price being ≈ 10.9 for new customers, and ≈ 14.7 for returning customers.

- setreward: Finally, after generating the reward, the summary step for this policy can be implemented as follows:

```
# Extract values:
# Create dummy for customer
if(self.context["customer"] == "returning"):
    customer = 1
else:
    customer = 0
price = self.action["price"]

# Create feature vector and response:
X = [1, price, price**2, customer, customer*price,
↪   customer*price**2]
y = self.reward["revenue"]

# Instantiate the m = 100 lm models
```

```
BTS = bts.BTS(self.get_theta(), lm.LM, m = 100, \
      default_params = {'b': np.zeros(6).tolist(), 'A' :
       ↪  np.identity(6).tolist(), 'n' : 0})

# Update the model parameters using the new observation
BTS.update(y, X)

# Store the updated values
self.set_theta(BTS)
```

To illustrate the outcomes of this sequential allocation scheme we run a simulation with $N = 1000$ and seed $= 43123$ setting the "log results" to True. Next, using the logged data, we plot the selected prices for each of the customer types separately. Figure 2.4 shows the progression of the recommended prices for each customer type; it is clear that these display a lot of exploration behavior early in the data stream, but after about 100 observations the BTS policy seems to exploit more and settles on a price that is close to the maximum in a large number of the interactions.

2.3.5 Lock-in Feedback

Picking a price was considered the intended action in the previous example. Hence, in this case $a_t \in \mathbb{R}$. This so-called continuum bandit problem (Bubeck, Munos, & Stoltz, 2011) has many practical applications. Here we provide an

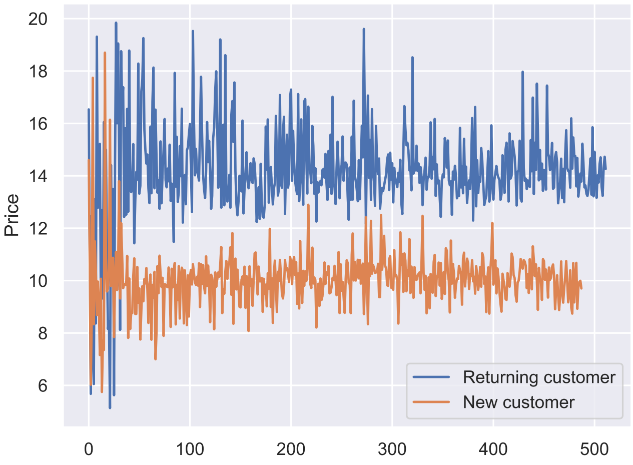


Figure 2.4: Overview of the selected prices of BTS with $m = 10$ and $N = 1000$ for both returning and new customers (separate lines).

example of an alternative strategy for selecting the actions in such a setting. The term "lock-in feedback" has been coined for this policy, which is described in detail in Kaptein and Ianuzzi (2016). The basic idea of the policy is to oscillate the values of the actions at a known frequency and to amplify this frequency in the observed rewards. Next, the noise can be integrated out, which produces a result that – given mild assumptions regarding the function relating the reward and the action, which we denote $r = f(a)$ – is directly proportional to the first derivative of $f()$. Subsequently, this first derivative can be applied, using a gradient-ascent-type algorithm, to move a step towards the maximum of $f()$.[2]

Lock-in Feedback is appealing because the experimenter does not need to specify $f()$ explicitly – as we did in the previous example – and the allocation policy has proved to be robust in cases of concept drift (e.g., a situation in which $f()$ changes over time). Lock-in Feedback can be implemented as follows:

- `getcontext`: For the sake of simplicity we consider a case without contextual information.

- `getaction`: The implementation of the decision step Lock-in Feedback is relatively simple using the **lif** library:

  ```
  theta = self.get_theta(all_float = False)
  Lif = lif.LiF(theta, x0 = 3.0, a = 0.5, t = 20, gamma = 0.02,
      omega = 1.0, lifversion = 1)
  suggestion = Lif.suggest()
  self.action["x"] = suggestion["x"]
  self.action["t"] = suggestion["t"]
  self.action["x0"] = suggestion["x0"]
  ```

  where we refer to the `lif` documentation at http://nth-iteration-labs.github.io/streamingbandit/ for details regarding the parameters of the `lif` method.

- `getreward`: Rewards can be simulated as follows:

  ```
  x = self.action["a"]
  self.reward["r"] = -1 * pow((x - 5), 2)
  ```

  where clearly the highest reward is obtained when $a = 5$.

- `setreward`: Finally, the summary step can be implemented using:

[2] Note, however, that Lock-in Feedback does not attain asymptotically optimal performance due to its constant exploration.

```
theta = self.get_theta(all_float = False)
Lif = lif.LiF(theta, x0 = 3.0, a = 0.5, t = 20, gamma = 0.02,
    omega = 1.0, lifversion = 1)
Lif.update(self.action["t"], self.action["x"],
↪   self.reward["r"],
    self.action["x0"])
self.set_theta(Lif)
```

Running a simulation with $n = 1000$ and `seed` $= 43123$ gives:

```
{
    "experiment": "2c070b0c17",
    "simulate": "success",
    "theta": {
        "x0": "4.9885573624026183",
        "t": "1000",
        "Yw":
        "[[981.0, 5.32735516395185, -0.0364325832561265],
        [982.0, 4.8625703653723935, 0.0023573471591685955],
        [983.0, 4.512596828979011, 0.11280696009422152],
        [984.0, 4.599313553861246, 0.06234374986452693],
        [985.0, 5.0430128577187805, -0.00010219719086955388],
        [986.0, 5.435840666435192, -0.0851018295535888],
        [987.0, 5.416689112014516, -0.07446606113542824],
        [988.0, 5.00321866842203, -1.599797646219837e-07],
        [989.0, 4.575643921281427, 0.07422661156312545],
        [990.0, 4.527110675559689, 0.10305907582127445],
        [991.0, 4.902338977243727, 0.0008184672724298606],
        [992.0, 5.356344978818639, -0.046745403226967665],
        [993.0, 5.471845324305438, -0.10767081744630806],
        [994.0, 5.142633922830094, -0.00314257165629084],
        [995.0, 4.671518435948291, 0.034171390252964014],
        [996.0, 4.491614635071014, 0.128372350733584],
        [997.0, 4.7684753555595965, 0.011794462261213126],
        [998.0, 5.247511963923335, -0.01586222111864476],
        [999.0, 5.488454341053454, -0.11925205127092639],
        [1000.0, 5.26974690054797, -0.020460304127878852]]"
    }
}
```

Where Yw and t are internals used to execute the policy, and x0 represents the current location of the search algorithm; initialized at 3.0 it is, with a value of 4.988 after $t = 1000$ observations, indeed close to the actual maximum of 5.

The **libs.lif** library has already been applied successfully in various settings, as described, for example, in a recent paper investigating the use of the LiF algorithm to optimize scenarios in behavioral economics (Kaptein, van Emden, & Iannuzzi, 2016b), and in another paper in which LiF is applied to the optimization of the physical features of an avatar in multiple dimensions in response to a continuous stream of ratings, provided by the participants of the experiment (Kaptein, van Emden, & Iannuzzi, 2016a). In both settings, LiF proved admirably capable of finding, and locking into, optima – despite the considerable noise often inherent in such human-choice-related studies. Hence, **StreamingBandit** was used successfully in these settings to allocate, in real-time, experimental treatments to subjects in a social-science study.

2.3.6 Nesting of policies

A further interesting use of **StreamingBandit** relates to the ability to nest multiple policies; this allows the user to, e.g., use an ϵ-greedy strategy to decide between the use of LiF and BTS, as presented above. Here we provide an example of this nesting of policies in which we assume that the user has instantiated two experiments, one implementing ϵ-first as described in Section 2.2, and one implementing ϵ-first as described in Section 2.3.1. We can now set up a third experiment that allocates interactions to either of these two experiments by referring to their exp_id's[3]. This can be achieved as follows:

- getcontext: We do not consider a context in this example
- getaction: Let us assume that we wish to uniformly randomly allocate half of our interactions to the ϵ-first experiment, and half of our interactions to the ϵ-greedy experiment. This can be done using:

```
id1 = "275fc0a66" # The exp_id of E-First
id2 = "18aec502c2" # The exp_id of E-Greedy

choice = np.random.binomial(1, 0.5)

# Run the e-first experiment
```

[3] Note that including a non-existent exp_id leads to errors in running the code. **StreamingBandit** does not explicitly check for such errors inside the code of the user. We have implemented an implicit call that can be used to check if the experiment is valid by using exp_nested.is_valid().

```
if choice == 0:
    exp_nested = Experiment(exp_id = id1)
    self.action = exp_nested.run_action_code(context = {})
    # We return the experiment number for later use
    self.action["experiment"] = id1
    # We re-compute the propensity based on the probability of
    ↪  picking the nested experiment
    self.action["propensity"] = self.action["propensity"] * 0.5
# or, run the e-greedy experiment
else:
    exp_nested = Experiment(exp_id = id2)
    self.action = exp_nested.run_action_code(context = {})
    self.action["experiment"] = id2
    self.action["propensity"] = self.action["propensity"] * 0.5
```

- `getreward`: Rewards can be simulated using the code we also introduced in Section 2.3.1.
- `setreward`: The summary step for these nested experiments can be implemented using:

```
# Based on the exp_id we know which experiment to update
exp_id = self.action["experiment"]

exp_nested = Experiment(exp_id = exp_id)
exp_nested.run_reward_code(context = self.context,
    action = self.action, reward = self.reward)
```

Which simply, based on the supplied `exp_id`, updates the correct experiment.

Running a simulation with $n = 2$ and `seed` $= 13214$, and the output set to `verbose`, gives:

```
{
    "data": {
        "0": {
            "theta": {},
            "context": {},
            "reward": {
                "value": 4.439687566610595
            },
            "action": {
                "propensity": 0.45,
                "experiment": "18aec502c2",
                "treatment": "treatment"
            }
        },
        "1": {
            "theta": {},
            "context": {},
            "reward": {
                "value": 7.559583564021055
            },
            "action": {
                "propensity": 0.25,
                "experiment": "275fc0a66",
                "treatment": "treatment"
            }
        }
    },
    "experiment": "1c57b6d641",
    "simulate": "success"
}
```

Which shows that in the first interaction ϵ-greedy was selected, which subsequently selected the treatment arm, and in the second interaction ϵ-first was selected. Obviously, this functionality can be greatly extended to use any sequential decision policy to decide between any other policy. This nesting makes **StreamingBandit** a versatile tool; we illustrate a practical application of the nesting in Section 2.3.7.

2.3.7 Parallel evaluation of multiple policies

Whereas the nesting discussed in the previous section allows one to allocate different interactions to different policies, the example we provide here allows one to evaluate, using a measure of average reward for example, multiple bandit policies in parallel. The idea behind the parallel evaluation derives from recent work on the offline evaluation of bandit policies. Li et al. (2011) show that one can evaluate multiple bandit policies offline by simply running through an existing data set of actions and rewards obtained using uniform random selections of the actions. For each interaction t in the offline data set one uses a bandit policy to generate a proposal action a'_t, and if the randomly selected action at that point in time matches the proposal (thus $a'_t = a_t$), then the reward is used to update the estimated performance of the policy. If not, then the time point is discarded. This leads to an evaluation of the policy with an expected number of observations of $\frac{1}{k}T$, where k is the number of possible actions and T the total number of observations in the offline data set. Multiple offline evaluation runs can subsequently be used to estimate and compare the expected performance of different policies.

Here we extend this idea to the parallel evaluation of multiple bandit policies. The implementation in **StreamingBandit** to compare, in parallel, the performance of the ϵ-first and ϵ-greedy experiments as introduced above is surprisingly straightforward:

- `getcontext`: For simplicity we again consider an empty context.
- `getaction`: In the decision step an action is chosen at random:

```
self.action["treatment"] =
↪  random.choice(["control","treatment"])
```

- `getreward`: Rewards can again be simulated using the code we also introduced in Section 2.3.1.
- `setreward`: Finally, after generating a reward, the summary step for the parallel evaluation of the policies is given below, where we again insert comments in the code to improve readability:

```
# Create a list of experiments / policies to evaluate
policies = ["18aec502c2", # E-Greedy
    "275fc0a66"]  # E-First

# For each experiment
for exp_id in policies:

    # Initialize the experiment:
    exp_nested = Experiment(exp_id)

    # Compute the suggested action:
    suggestion = exp_nested.run_action_code(context = {})

    # See if the suggested action matches the actual action:
    if suggestion["treatment"] == self.action["treatment"]:
        # And if so store the performance of the policy:
        mean = base.Mean(self.get_theta(key = "policy_means",
            value = exp_id))
        mean.update(self.reward["value"])
        self.set_theta(mean, key = "policy_means", value =
        ↪   exp_id)

        # And finally update the policy:
        exp_nested.run_reward_code(context = {},
            action = self.action, reward = self.reward)
```

This code implements Algorithm 2 of Li et al., 2011.

Running a simulation with $n = 250$ and `seed` $= 43123$ using the above specification gives:

```
{
    "theta": {
        "policy_means:275fc0a66": {
            "m": "5.243151928222057",
            "n": "114"
        },
        "policy_means:18aec502c2": {
            "m": "6.050783848360902",
            "n": "114"
        }
    },
    "experiment": "270ed59474",
    "simulate": "success"
}
```

This output shows that, in this test run, the average reward of the ϵ-greedy policy is slightly higher than that of the ϵ-first policy. This is due to the fact that ϵ-first has a random exploration phase of $n = 100$. Since both policies now only have had 114 accepted actions, ϵ-first will have explored much more than ϵ-greedy and will choose the suboptimal action more, resulting in a lower average reward.

2.4 Applied usage

In this section, we describe some of the practical applications of **StreamingBandit**. First, we explore its use in assessing the effects of discounts in online selling; this small, initial trial highlights the simple use of **StreamingBandit** to collect data in-the-field. Second, we introduce its use in a social-science experiment.

2.4.1 Online marketing

StreamingBandit was used by an online cash-refund company to examine the effects of their pricing scheme. The company offers customers the opportunity to sign up for a refund program. After signing up they are provided with discounts, in the form of a cash refund, as long as their online purchases are carried out through the online platform. The refund company has negotiated different agreements with a large number of different e-commerce stores, and the discount percentages they have obtained vary from store to store. By default, the refund company offers half of its negotiated discount to the customer, and takes the other half as a fee for its services. However, it has no clear idea as to whether this 50/50 (or $\frac{1}{2}$) split is optimal in the sense that it maximizes its profit, which is influenced by the total number of purchases, the size of the purchases, and the way in which the negotiated discount is split between the company and the customer.

The company set up **StreamingBandit** to explore the effects of the different splits – in their definition running from 0 to 1 where 1 means that the total negotiated discount is fully passed on to the customer and 0 means that all of it is retained by the company – on their resulting profits. Here we present a simple implementation of the random exploration of different splits that the company carried out for a very small number of $n = 103$ unique customers in one specific store. The implementation was as follows:

- `getcontext`: Because this is a field exploration, the context was provided by the participating company. It consisted of a JSON object containing the `maxpercentage`, which contained the negotiated discount for the specific store that was viewed by a customer. It looked like this:

  ```
  {"context" : {"maxpercentage" : 8.5}}
  ```

  where the `maxpercentage` for the specific store from which our presented data originated was always 8.5%. However, our implementation described below is able to address changing maximum percentage(s) between different stores. Note that this can be simulated in **StreamingBandit** using the following `getcontext` code:

  ```
  self.context["maxpercentage"] = numpy.random.uniform(1,10)
  ```

- `getaction`: The implementation of the decision step was straightforward since the company initially set out merely to examine the effects of random fluctuations of the discounts offered. The implementation was as follows:

  ```
  maxpercentage = self.context['maxpercentage']
  split = np.random.uniform()
  discount = split * maxpercentage
  self.action['split'] = split
  self.action['discount'] = discount
  ```

  Here, first the `maxpercentage` is retrieved. Next, a split is computed with `split` $\sim$ `unif`$(0, 1)$, after which the percentage discount to be offered to the customer is computed and then both the `split` and the actual `discount` are returned in the `action` object.

- `getreward`: The online platform would display the computed `discount` to the visiting customer, and subsequently a reward would be generated by virtue of the customer's purchasing one or multiple products resulting in a `revenue`. The online platform returns both the `revenue` as well as the `split` and `discount`. This can be simulated using:

  ```
  self.action['split'] = self.action['split']
  self.action['discount'] = self.action['discount']
  self.reward['revenue'] = numpy.random.uniform(0,100)
  ```

- `setreward`: Finally, given that the aim of the company was merely to collect data on the effect of the changing splits, it did not need any `setreward` code because **StreamingBandit** automatically logs all the data that is received with a `setreward` call.

This simple implementation allowed the refund company to vary the split randomly (instead of using the current de-facto $\frac{1}{2}$ split) and to log the resulting revenue.

Figure 2.5 provides an overview of the relation between the suggested `split` and the resulting profit in euros of the refund company. The `profit` for the rebate company is defined as the maximum discount percentage (8.5%) times one minus the split (between 0 and 1), times the revenue. Each dot represents one completed purchase by one customer (possibly containing multiple products). Note that while we limit the presented results here to a single e-commerce store, the store sells multiple products and hence the revenue per customer can vary greatly. It seems from the limited data of these $n = 103$ unique customers for a single store

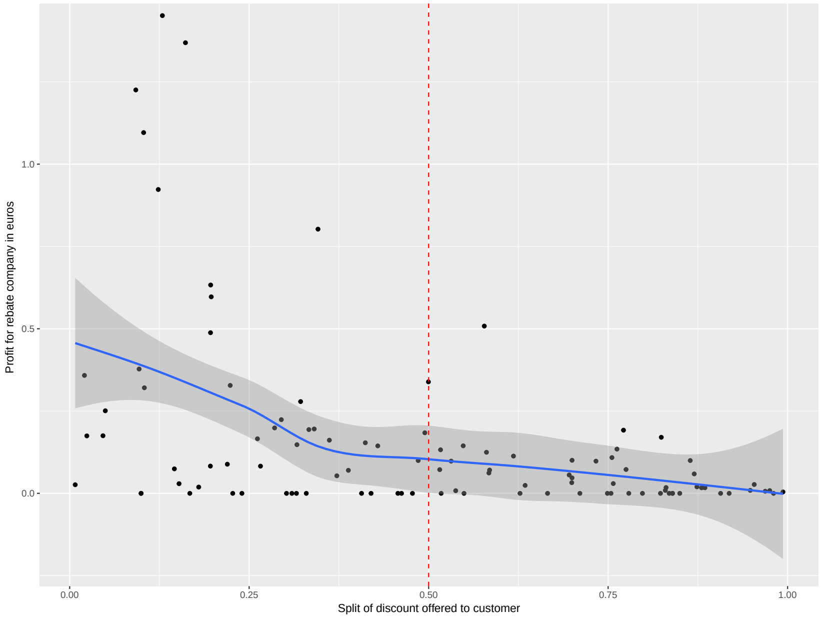


Figure 2.5: Overview of the effect of the offered split of the discount on the profit of the refund company in euros. Data collected using random selection of the refund percentage using **StreamingBandit**. The figure presents data on $n = 103$ unique customers. The dashed red line represents the company's current $\frac{1}{2}$ split.

that a high customer-refund offer – but as a result a low margin for the company – leads to low profits, whereas an offer that is significantly below the current $\frac{1}{2}$ split increases the company's profits.

The company intends to use **StreamingBandit**, now that the software is integrated into its current online service, to experiment with different sequential allocation schemes that offer different splits between competing stores or between different customers. Using the random data and an adaption of the offline evaluation method developed by Li et al., 2011 (also described in Section 2.3.7), the company hopes to find the policy that has the best model fit on their data. Note that here every step towards solving this statistical decision problem involves using **StreamingBandit** – from gathering data, to policy evaluation, to the final, live setting. This provides a simple example of the utility of **StreamingBandit** for field trials of bandit policies.

2.4.2 Social science experiment

The second applied use of **StreamingBandit** we present concerns a social-science experiment examining the decoy effect (see Kaptein, van Emden, & Iannuzzi, 2016b, for a full description of the experiment). In short, the decoy effect states that people may be persuaded to switch from one offer to another by the presence of a third option (the decoy) that, rationally, should have no influence on the decision-making process. For example, when asked to choose between a laptop with a good battery but a poor memory and a laptop with a poor battery but a good memory, people seem to shift their preference between the two if the offer is accompanied by a third laptop, the decoy, that has a battery as good as the latter but an even worse memory, and hence should in any case be an irrelevant option. The placement of the decoy in the product-attribute space is heavily studied in the literature: researchers manipulate the exact battery life in hours and the RAM in GB of the decoy laptop, and study the resulting choices that people make.

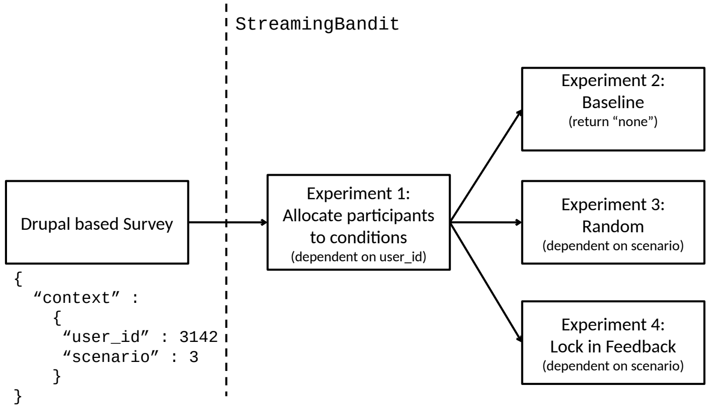
Kaptein, van Emden, and Iannuzzi (2016b) used **StreamingBandit** to study whether Lock-in Feedback, the sequential optimization scheme introduced in Section 2.3.5, can be used to find the optimal placement of the decoy – only considering changes on one dimension. The authors considered not only the laptop scenario but also eight different decoy scenarios. The study was carried out online using a `drupal`-based survey, which communicated with **StreamingBandit** to implement the allocation of the exact positioning of the decoy. The researchers allocated participants to one of 3 between-subject conditions using **StreamingBandit**:

1. Baseline: participants in this condition were presented with a binary choice between two products, and no decoy was present. This was implemented by sending an `action` with `{"decoy":"none"}` response to the survey front-end.

2. Random: participants in this condition were presented with a random positioning of the decoy. The range of possible values of the random positioning depended on the specific scenario, and were hard-coded and retrieved using the `scenario` supplied in the context.

3. Lock-in Feedback: participants in this condition were presented with a value of the decoy that depended on the previous interactions of other participants. The Lock-in Feedback algorithm was used to suggest a new placement each time a participant viewed a product. Subsequently, the (binary) choice made by the participant was used to update the algorithm in the `setreward` stage. We refer the reader to (Kaptein, van Emden, & Iannuzzi, 2016b) for details and for the exact settings of the tuning parameters.

Figure 2.6 presents an overview of the setup of this study. A number of the details of the implementation are covered in earlier sections of this paper: the implementation of both the baseline and the random condition are straightforward, with `self.action["decoy"] = "none"` and `self.action["decoy"] = np.random.uniform(low,high)`, respectively, as the core `getaction` implementations. In



```
{
  "context" :
    {
     "user_id" : 3142
     "scenario" : 3
    }
}
```

Figure 2.6: Schematic setup of the 4 **StreamingBandit** experiments used to realize the data-collection in (Kaptein, van Emden, & Iannuzzi, 2016b).

the latter implementation the `low` and `high` bounds were implemented as a simple list indexed by the `scenario` number. Finally, the Lock-in Feedback condition was implemented as presented in Section 2.3.5, the only exception being that the `theta` was stored independently for each `scenario`. Hence, the novel part of the implementation of this study is the persistent allocation of participants to one of the three conditions; this was achieved in experiment 1 in Figure 2.6 by using the following `getaction` code:

```
if not("condition" in self.get_theta("user_id",
↪   self.context["user_id"])):
    self.action["note"] = "First allocation"
    draw = random.choice(["baseline", "random", "lockin"])
    self.set_theta({"condition":draw}, "userid",
     ↪   self.context["userid"])

self.action["condition"] =
    self.get_theta("userid", self.context["userid"])["condition"]
```

which assigns participants randomly to one of the three conditions persistently based on the `user_id` supplied in the context to the `getaction` call.[4] The data resulting from this experiment are available at http://dx.doi.org/10.7910/DVN/FCHU0J. This field implementation provides a prime example of the use of

[4] Note that the actual implementation in the study differed slightly to allow for unequal sample sizes in each of the conditions. In addition, the baseline and random conditions where manually removed after sufficient data had been collected.

StreamingBandit, both for the allocation of participants to conditions in (web-based) experiments, as well as for the use of sequential decision policies such as Lock-in Feedback in such experiments.

2.5 Conclusion and future work

This paper presented **StreamingBandit**, a RESTful web application that enables researchers to develop, evaluate, and deploy CMAB policies in online experiments and field studies. By making **StreamingBandit** publicly available we hope to contribute to the more extensive use of such policies to solve statistical decision problems. The software could help in extending the currently prevailing use of basic random assignment to the use of more refined strategies throughout the social and medical sciences. To that effect, we started out with a clarification of the design rationale behind **StreamingBandit**. We explained our decision to split up the summary and the decision step of a policy – a split meant to encourage the implementation of computationally efficient online policies. We subsequently illustrated **StreamingBandit**'s versatility and flexibility in a number of examples, and we concluded with two case studies in which we used **StreamingBandit** to run field experiments.

We are currently aware of a number of limitations of **StreamingBandit**. First, as of now, **StreamingBandit** still runs single-threaded. Although parallelization for larger-scale applications ought to be relatively easy to implement on the level of policies, it may prove substantially harder within policies. Nevertheless, by forcing policies online by design, and using state-of-the-art web technology for its back-end, **StreamingBandit** is already more than capable of being deployed in a multitude of small-to-medium-sized field trials. We are of the opinion that parallelization is an obvious next step in **StreamingBandit**'s development, ensuring its future scalability.

Second, in some applications we find that certain types of rewards manifest themselves faster than others. In one instance of the use of **StreamingBandit**, for example, the decision to reject a loan to a customer after an application had been submitted to the firm was much faster than the decision (and subsequent confirmation) to accept the customer. Such an asymmetric delay might bias learning and thus needs to be addressed. Currently, we do not provide an off-the-shelf solution to this problem – admittedly because it is thus far unclear to us how to address the problem in general – hence users will need to resort to custom implementations of the `getaction` and `setreward` codes to deal with this issue.

Finally, our current CMAB libraries and toolkit still offer ample room for improvement and extension. Outside of the currently implemented methods, there are many more policies that address the exploration-exploitation trade-off in various settings. In that respect, we hope and expect the open-source nature of **StreamingBandit** to be conducive to the continued growth of the platform, encouraging researchers to implement, test, and disseminate new and existing bandit policies and algorithms.

2.A Setting up an experiment

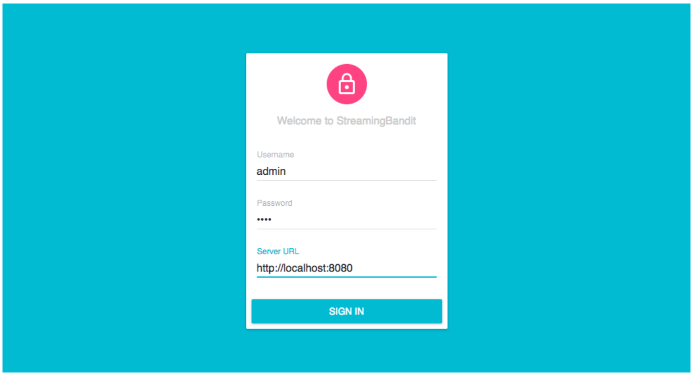
This appendix introduces the front-end of **StreamingBandit**.[5] We will how show to get from the login screen to setting up your first simulation using one of the default experiments.

First, when you have set up the front-end (using, e.g., the available **Docker** container), go to the login screen in your browser (for the **Docker** container this would be http://localhost or the Docker-set IP address) as shown in Figure 2.7.

After logging in, you will find the dashboard as in Figure 2.8. To show all the active experiments, click on `Experiments`. This will bring you to an environment as shown in Figure 2.10. Continue clicking on the `Create` button, which will give you an empty `Create Experiment` field, as in Figure 2.10.

On the creation page you can fill in a name, for example `E-First`, and select a default experiment from the `Use experiment template` list. Selecting the default ϵ-first experiment, will end up with a filled-in form, as in Figure 2.11. Next, clicking on the `Save` button will save the experiment in the database.

When the experiment has been created, the dashboard (Figure 2.12) shows that the experiment is active and has an ID and key assigned. Clicking on the `Edit` button will take you back to the settings of the experiment. Now you can choose to go to the `Simulate` tab as displayed in Figure 2.13. After filling in 1000 for the number of iterations and 43123 as the seed you can click `Run a simulation of the experiment`, which will give a result as in Figure 2.14. Finally, you can click on the `Theta` tab and inspect the parameters that are stored in the database (Figure 2.15). Here you can also download the data that has been logged for the current experiment.



Welcome to StreamingBandit

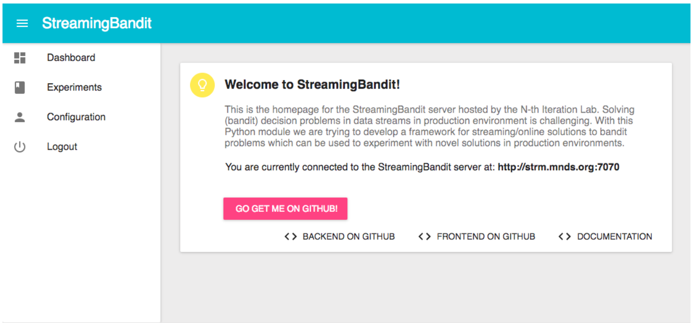
Username
admin

Password
····

Server URL
http://localhost:8080

Figure 2.7: The login screen. Here you can set the URL for the back-end and login.

[5] Which can be found at https://github.com/Nth-iteration-labs/streamingbandit-ui.

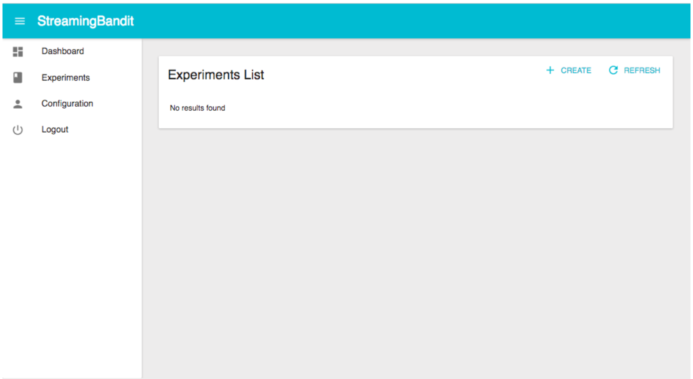


Welcome to StreamingBandit!

This is the homepage for the StreamingBandit server hosted by the N-th Iteration Lab. Solving (bandit) decision problems in data streams in production environment is challenging. With this Python module we are trying to develop a framework for streaming/online solutions to bandit problems which can be used to experiment with novel solutions in production environments.

You are currently connected to the StreamingBandit server at: **http://strm.mnds.org:7070**

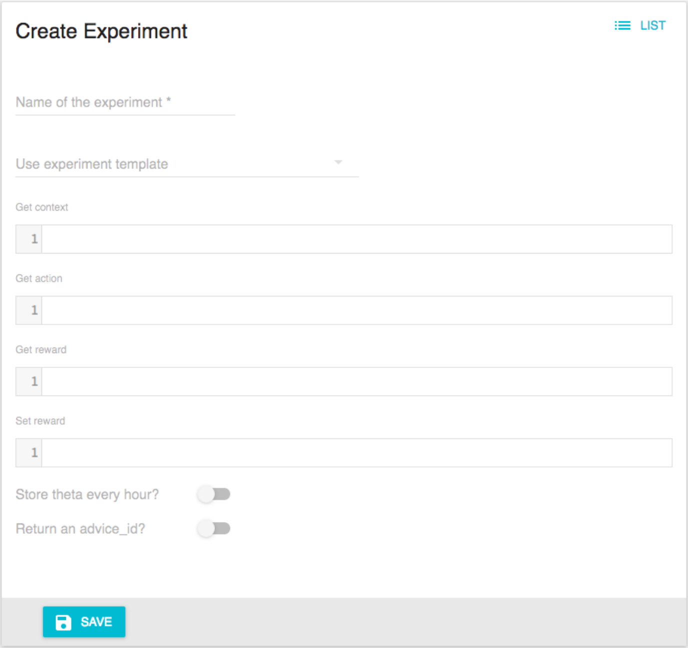
Figure 2.8: The dashboard of **StreamingBandit**, here you can navigate to the list of experiments and find some extra information.



Experiments List

No results found

Figure 2.9: This screenshot shows an empty list of experiments. You can start creating an experiment by clicking on the `Create` button.



Create Experiment

Name of the experiment *

Use experiment template

Get context

```
1
```

Get action

```
1
```

Get reward

```
1
```

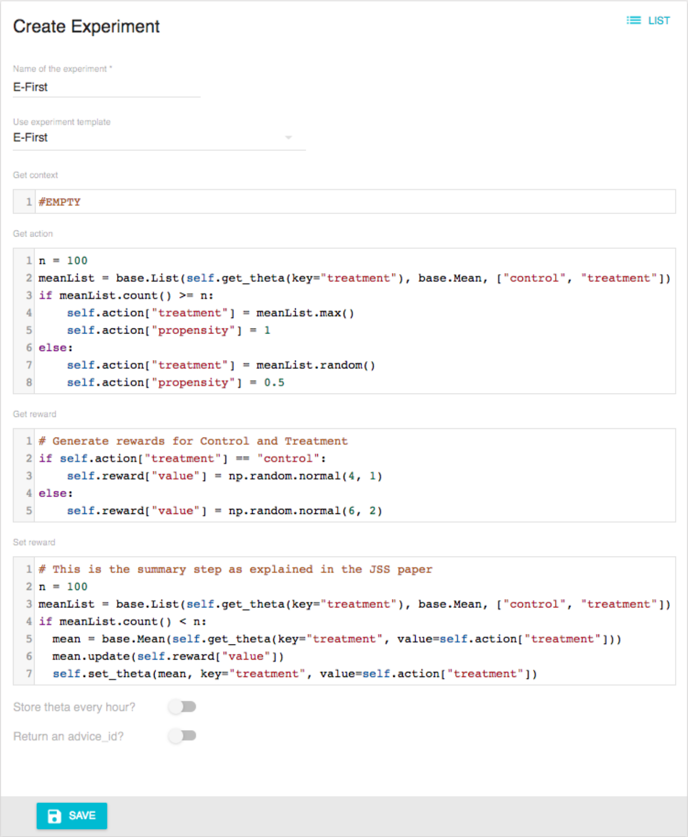
Set reward

```
1
```

Store theta every hour?

Return an advice_id?

Figure 2.10: This is an empty form for an experiment, which is normally shown inside the dashboard. Here you can type a name, choose a default and edit the code for the experiment.



Create Experiment

Name of the experiment *
E-First

Use experiment template
E-First

Get context

```
#EMPTY
```

Get action

```
n = 100
meanList = base.List(self.get_theta(key="treatment"), base.Mean, ["control", "treatment"])
if meanList.count() >= n:
    self.action["treatment"] = meanList.max()
    self.action["propensity"] = 1
else:
    self.action["treatment"] = meanList.random()
    self.action["propensity"] = 0.5
```

Get reward

```
# Generate rewards for Control and Treatment
if self.action["treatment"] == "control":
    self.reward["value"] = np.random.normal(4, 1)
else:
    self.reward["value"] = np.random.normal(6, 2)
```

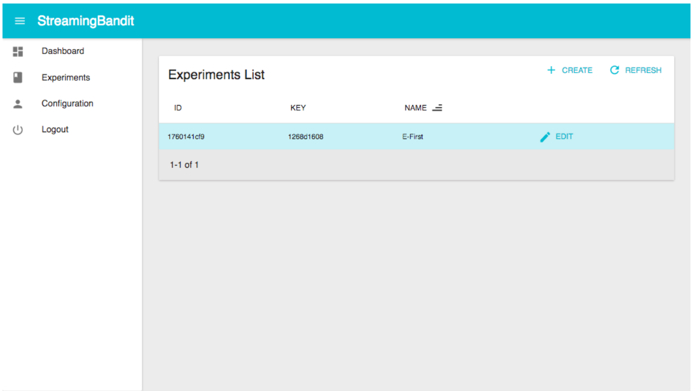
Set reward

```
# This is the summary step as explained in the JSS paper
n = 100
meanList = base.List(self.get_theta(key="treatment"), base.Mean, ["control", "treatment"])
if meanList.count() < n:
  mean = base.Mean(self.get_theta(key="treatment", value=self.action["treatment"]))
  mean.update(self.reward["value"])
  self.set_theta(mean, key="treatment", value=self.action["treatment"])
```

Store theta every hour?

Return an advice_id?

Figure 2.11: We have selected the E-First template, which is automatically filled in the correct fields. Pressing the Save button will create the experiment.

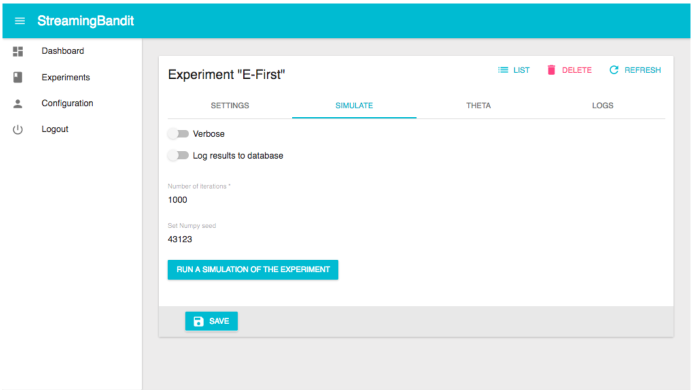


Experiments List

ID	KEY	NAME
1760141cf9	1268d1608	E-First

1-1 of 1

Figure 2.12: If you return to the dashboard you can view and edit your newly created experiment (and the associated ID and key).



Experiment "E-First"

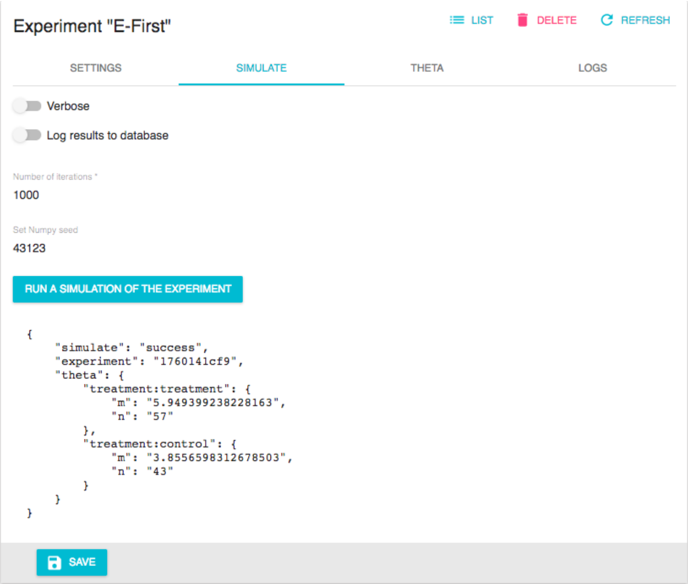
Verbose

Log results to database

Number of iterations *
1000

Set Numpy seed
43123

Figure 2.13: Here you can see the simulation panel, which you can use to easily run a simulation of the experiment. You can set the seed and the the number of iterations, log the results to the database and even show verbose results. Click `Run a simulation of the experiment` to run a simulation and get an output of the results.



Experiment "E-First"

Verbose

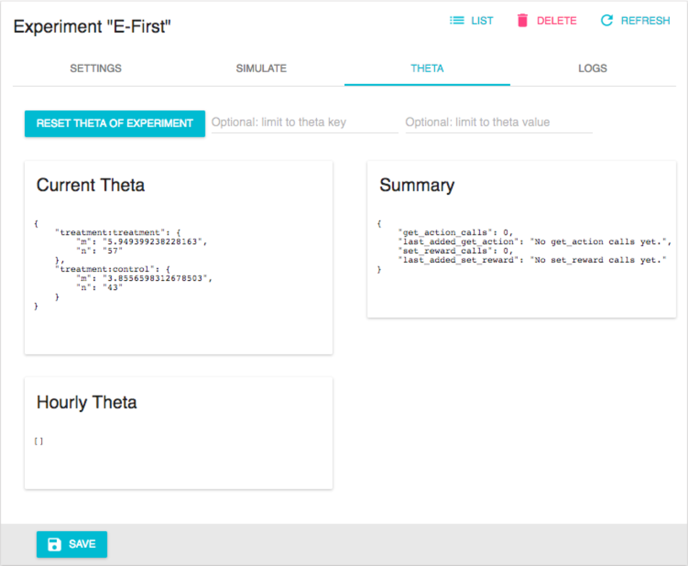
Log results to database

Number of iterations *
1000

Set Numpy seed
43123

```
{
    "simulate": "success",
    "experiment": "1760141cf9",
    "theta": {
        "treatment:treatment": {
            "m": "5.949399238228163",
            "n": "57"
        },
        "treatment:control": {
            "m": "3.8556598312678503",
            "n": "43"
        }
    }
}
```

Figure 2.14: Run a simulation and look at the output of the results.



Experiment "E-First"

Optional: limit to theta key

Optional: limit to theta value

Current Theta

```
{
    "treatment:treatment": {
        "m": "5.949399238228163",
        "n": "57"
    },
    "treatment:control": {
        "m": "3.8556598312678503",
        "n": "43"
    }
}
```

Summary

```
{
    "get_action_calls": 0,
    "last_added_get_action": "No get_action calls yet.",
    "set_reward_calls": 0,
    "last_added_set_reward": "No set_reward calls yet."
}
```

Hourly Theta

```
[]
```

Figure 2.15: The theta panel shows the current state of θ and other information.

Chapter 3

Exploiting Nested Data Structures in Multi-Armed Bandits

Abstract

The multi-armed bandit (MAB) problem provides a formalization for sequential decision-making. This theoretical problem formalization has many real-world applications: e.g., choosing between different medical treatments or choosing between different online advertisements. However, in these real-world applications we often find that there is some form of nested (or hierarchical) structure, e.g., we observe multiple observations from the same patient or from the same user. This hierarchical structure is all too often ignored in the current MAB literature. In this paper we introduce means of exploiting hierarchical structures via so-called partial pooling (or shrinkage) methods, and we adapt a number of popular MAB policies to incorporate this idea. We focus specifically on the Bernoulli MAB problem. Through extensive simulations and an empirical evaluation we show that our proposal improves the performance of a number of popular policies.

Keywords: multi-armed bandit, dependent observations, shrinkage factors, hierarchical Bayesian modeling

Submitted as Kruijswijk, J.M.A., Böing-Messing, F., Van Emden, R., Kaptein, M.C. Exploiting Nested Data Structures in Multi-Armed Bandits.

3.1 Introduction

In recent years researchers in various scientific disciplines are increasingly investigating the use of sequential treatment allocation methods when designing experiments (see e.g., Bouneffouf & Rish, 2019; Eckles & Kaptein, 2019; Kaptein, 2014; Press, 2009)). For example, Kaptein et al. (2017), applied sequential allocation methods to study social phenomena (in this case, the perception of beauty) that are hard to study due to natural occuring effects such as treatment heterogeneity and measurement error. Furthermore, in Clement, Roy, Oudeyer, and Lopes (2015), the authors used sequential experimentation to adaptively personalize learning activities in a tutoring system to maximize skill acquisition for students and Kaptein, van Emden, and Iannuzzi (2016b) implemented a sequential treatment allocation scheme to investigate the limits of the decoy effect – a well known human bias.

The design of sequential experiments can often be formalized as a multi-armed bandit (MAB) problem, in which (in its canonical form) a gambler stands in front of a row of slot machines, each with a (potentially) different payoff (Berry & Fristedt, 1985; Whittle, 1980). The gambler has to sequentially decide which slot machine to play, such that he makes as much profit as possible in the long run (expressed in terms of cumulative reward). In the beginning of the process the gambler has no knowledge of the different payoffs of the machines. At each time point, he can choose to gain more knowledge regarding a machine he is uncertain about (exploration), or gear his choice towards exploiting his current knowledge and play the machine with the highest expected payoff (exploitation) (Berry & Fristedt, 1985). A good decision strategy – or *policy* – balances this so called exploration-exploitation trade-off and does not waste too many plays on gaining new knowledge, nor does it become too greedy and get stuck exploiting a suboptimal machine (Kaelbling, Littman, & Moore, 1996).

The canonical MAB problem has been heavily researched (Berry & Fristedt, 1985; Hardwick, Oehmke, & Stout, 1999; Robbins, 1952; Scott, 2010), and a large number of policies have been proposed and evaluated. These include the popular and asymptotically optimal Thompson sampling (Agrawal & Goyal, 2012, 2013a, 2013b; Chapelle & Li, 2011; Scott, 2010; Thompson, 1933), Upper Confidence Bound (UCB) methods (Auer, Cesa-Bianchi, & Fischer, 2002; Lai & Robbins, 1985), and more heuristically motivated policies such as ϵ-greedy, ϵ-first and Softmax (Sutton & Barto, 2011). More recently, a number of generalizations of the MAB problem have been proposed, such as the popular bandit problems with side observations (also called the contextual MAB problem) (Beygelzimer, Langford, Li, Reyzin, & Schapire, 2011; Bubeck & Cesa-Bianchi, 2012; Langford & Zhang, 2008; Wang, Kulkarni, & Poor, 2005). In the contextual MAB (CMAB) problem, the gambler first observes the current state of the world (context), which he can use to aid his decision for the next action. The (C)MAB problem has been used in many different situations, such as the personalization of online news (Li et al., 2010), online selling (Kaptein, McFarland, & Parvinen, 2018), website morphing (Hauser, Urban, Liberali, & Braun, 2009), and adaptive clinical trials (Press, 2009; Williamson, Jacko, Villar, & Jaki, 2017).

The CMAB problem tackles situations where it is expected that certain features of the context potentially influence the distribution of the rewards. For example, when personalizing online news items, we might observe users with the same interests or traits multiple times. In such cases it is likely that within a specific cluster of observations – those observations that share feature values – the outcomes are correlated. Examples include, but are not limited to, the repeated observations of individual click-through behavior on website advertisements (Cheng & Cantú-Paz, 2010), and the smoking behavior of students that are grouped within different schools (Murnaghan, Sihvonen, Leatherdale, & Kekki, 2007). In each of these situations we expect the observations within a cluster to be more similar than between clusters. However, contrary to many current CMAB approaches it is also reasonable to expect that even between clusters we can effectively "borrow strength" (Gelman & Hill, 2006): results obtained in one cluster might be helpful in understanding those in another cluster.

The potential intricate dependency of observations both within and between clusters is, in the social sciences, often addressed using hierarchical (or mixed) models. Prior work studying dependencies in (C)MAB problems has focussed on clustering of dependent arms (Pandey, Chakrabarti, & Agarwal, 2007) and taxonomy induced dependencies (Pandey, Agarwal, Chakrabarti, & Josifovski, 2007). However, surprisingly, none of these works explicitly focussed on modeling hierarchical dependencies in ways common to the social sciences. A seemingly appealing alternative close to our current work was provided in the case of personalization of online news (Li et al., 2010): the authors use a hybrid linear model that is based on a linear combination of an estimator common for all actions and an estimator that is action-specific (the policy is called LinUCB). In the case of LinUCB, however, a feature vector must be explicitly constructed *a priori* and the model is only defined after observing the reward of each action for each unique value of the feature vector (i.e. for each cluster) – or alternatively, by using strong prior information. As a consequence if, for example, you are modeling individual users, the default LinUCB approach will be effectively undefined when a new, unique user comes in. Furthermore, the LinUCB approach will result in a very large model (i.e., one with a large number of parameters) if there are many, potentially unique, users. Finally, there is no information sharing amongst the users. In such scenarios LinUCB thus does not allow us to adequately model the hierarchical structure present in the data.

3.1.1 Within and between cluster dependencies

Effectively, in the (non-contextual) MAB literature, any potential dependency within a cluster is fully ignored and observations are deemed independent (an approach referred to as *complete pooling*). In the CMAB literature it is common to model the outcomes within a cluster, but ignore dependencies between clusters. Thus, in the latter case, conditional on the cluster membership, the observations are independent (called *no pooling*). In this paper, we consider hybrid policies that model dependencies within and between clusters; this approach is called *partial pooling* (Gelman & Hill, 2006).

As an example, suppose we have two types of advertisements which we can serve to users that return to a webpage, possibly multiple times. On the one hand, we can choose to model the effect of the ad, ignoring the dependencies within returning customers. Conversely, we can choose to model the ad-person combinations; in such cases each ad-person combination is treated as distinct. Assuming customer heterogeneity, the no pooling analysis – treating each ad-person combination independently – seems the most obvious option. The problem that arises here is, however, that the number of observations within a user is often low, which leads to a poor estimate of the expected reward for an ad. Thus, while the no pooling approach takes into account the information of a hierarchical structure (observations within persons), this approach suffers from limited data at the lowest level of observations. On the other hand, one can choose to pick the complete pooling approach to solve the problem of low number of observations, but this approach completely ignores possible meaningful heterogeneity between persons; effectively, the complete pooling approach treats all users as equal (e.g., having the same success probabilities). Both approaches are in this case suboptimal.

The idea behind partial pooling is to introduce a compromise between the two extremes of complete pooling and no pooling (Efron & Morris, 1975; Gelman & Hill, 2006). This is done using some form of weighted average that reflects the amount of information that is present amongst all users and the observations within a user. If there are little to no observations available from a user, the weighting pulls the estimate of that user closer to the overall estimate; effectively this yields the complete pooling estimate. If there is a larger amount of observations available from a single user, the weighting will, in the limit, ensure that the estimate will be the user average; effectively this yields the no pooling estimate. When observations arrive sequentially, the estimates obtained for a single user will gradually shift between those two extremes. To illustrate why this partial pooling is useful, we can look at the difference between no and complete pooling. The no pooling variant has high variance as there are limited number of observations. The complete pooling variant is biased as it does not make a distinction between users. Partial pooling proves to be a better balance between the bias and variance in the estimator (James & Stein, 1961; Stein, 1956).

3.1.2 Overview

In the current article, we focus on extending existing MAB policies for the Bernoulli MAB problem to include a partial pooling approach to exploit the occurrence of hierarchical structures in sequential experiments. We do this by modeling the hierarchical structure using random effects as opposed to fixed effects (the common approach in the CMAB literature and effectively the approach taken in the LinUCB case). The objective of our study is to provide a contribution to the MAB literature by demonstrating the importance of hierarchical structures and providing an effective method for dealing with such structures.

The remainder of this paper is organized as follows. In the next section, we will give a formal introduction of the CMAB problem and a formal definition of the hierarchy structures that we are studying. Then, we will introduce and extend a number of existing MAB policies such that they include a hybrid approach. After that we conduct a simulation study to compare traditional MAB policies with their extended, partial pooling, versions – we also study a fully Bayesian hierarchical modeling approach. Furthermore, we conduct an empirical study using offline policy evaluation to further test the policies. Finally, we will discuss the results and discuss avenues for further research.

3.2 The contextual multi-armed bandit problem and nested data structures

The CMAB problem can formally be defined as follows: at each time $t = 1, ..., T$, we observe a context $x_t \in \mathcal{X}$. After we choose action $a_t \in \mathcal{A}$, we subsequently observe reward r_t from an unknown probability distribution $P(r|a, x)$. The aim is to find a *policy* Π – which is a mapping from all the historical data $\mathcal{D}$ (which contains all previous (x, a, r) triplets) and the current context x_t to the next action (a_t) – that selects actions such that the cumulative reward $R_c = \sum_{t=1}^{T} r_t$ is as large as possible.

To assess how a policy performs we often look at the expected cumulative regret of the policy instead of the cumulative reward, which is defined by:

$$\mathbb{E}[R_T] = \mathbb{E}\left[\sum_{t=1}^{T} r_t^* - r_t\right] \tag{3.1}$$

where r_t^* is the reward of the action with the highest expected reward (or equivalently, the reward of an oracle) and the expectation is taken over the randomness of the environment (i.e. the distribution of the reward) and the policy.

In the next subsections we introduce the data structure as it occurs with the canonical MAB problem schematically and subsequently introduce the nested data structure that we focus on in this paper.

3.2.1 Data structure for the canonical MAB problem

Figure 3.1 graphically depicts a traditional MAB setting for a K-armed bandit ($\mathcal{A} = \{1, \dots, K\}$). The arms $k = 1, \dots, K$ give rise to i.i.d. observations $r_1^k, \dots, r_{n^k}^k \sim \mathrm{P}(\theta^k)$ with n^k the number of observations for arm k. Note that in this paper we consider the rewards to be i.i.d. Bernoulli, and thus the parameter θ for each distribution is equal to its expected value. Hence, to minimize the expected regret, a policy needs to balance the exploration and exploitation of all arms based on the estimates of each θ^k and play the arm with the highest expected value as much as possible.

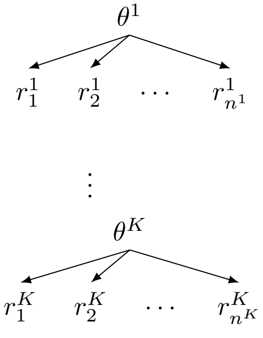


Figure 3.1: Graphical representation of the simple K-armed Bernoulli bandit problem. θ^1 through θ^K are the parameters for the reward distribution of each arm, which give rise to each reward r_1^K through $r_{n^K}^K$ (for arm K).

3.2.2 Data structure for the CMAB problem: potential nesting

Figure 3.2 schematically depicts the CMAB setting for a K-armed bandit in which a nested data structure is introduced which we study in this paper. In this case, we assume a context feature with options (e.g., users) $j = 1, \dots, J$ (with the interactions per user running from $i = 1, \dots, n_j^k$ for arm k) for which the distributions $\mathrm{P}(\theta_j^k)$ might be user and arm specific. However, we do not assume that the parameters of these different distributions are independent: rather, the θ_j's themselves originate from some distribution with mean θ (i.e., $\theta_j^k \sim \mathrm{P}(\theta^k)$). Hence, we are effectively dealing with a contextual bandit problem – where j identifies the user and functions as a context – in which there is a correlation within the clusters as well as between clusters. This situation is of particular interest when, say, θ^1 is higher than θ^2, but due to high variance in the distribution of the θ_j's this is the other way around for some users (i.e., $\exists j, j' : \theta_j^1 < \theta_j^2 \wedge \theta_{j'}^1 > \theta_{j'}^2$).

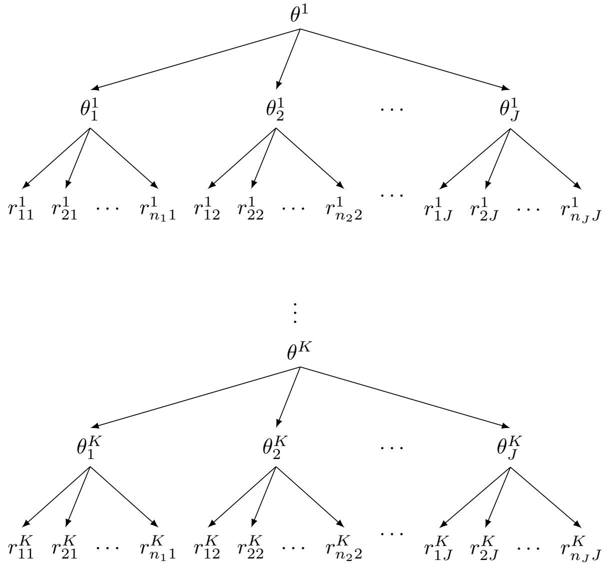

θ^1
θ_1^1
θ_2^1
$\cdots$
θ_J^1
r_{11}^1 r_{21}^1 $\cdots$ $r_{n_11}^1$ r_{12}^1 r_{22}^1 $\cdots$ $r_{n_22}^1$ $\cdots$ r_{1J}^1 r_{2J}^1 $\cdots$ $r_{n_JJ}^1$
$\vdots$
θ^K
θ_1^K
θ_2^K
$\cdots$
θ_J^K
r_{11}^K r_{21}^K $\cdots$ $r_{n_11}^K$ r_{12}^K r_{22}^K $\cdots$ $r_{n_22}^K$ $\cdots$ r_{1J}^K r_{2J}^K $\cdots$ $r_{n_JJ}^K$

Figure 3.2: Graphical representation of the contextual K-armed bandit problem. θ^1 through θ^K are the parameters for the reward distribution of each arm, which give rise to an individual parameter per user θ_1^K through θ_J^K. These in turn lead to each reward r_{1J}^K through $r_{n_JJ}^K$ (for user 1 and arm K with n_J the number of observations n for user J).

3.2.3 Dealing with nested data structures

Various options exist that would allow a MAB policy to deal with nested data. The easiest options to deal with the nested data generating scheme described in Figure 3.2 are to either ignore the nested structure (taking a complete pooling approach), or to treat each context j separately (effectively taking a no pooling approach). Choosing a hybrid approach is not trivial, as many different approaches to dealing with nested data structures have been suggested in the literature (Efron & Morris, 1975; Gelman & Hill, 2006). To demonstrate the benefits of adopting a hybrid approach, we focus on a simple implementation using shrinkage factors (Ippel et al., 2019; James & Stein, 1961) (i.e., a parameter that explicitly combines data at various levels in the hierarchy). Next, we will introduce a fully Bayesian way to deal with hierarchical dependencies. Thus, to clarify the suggested approach conceptually, we first provide a simple example of complete, no, and partial pooling:

1. In the complete pooling situation, one assumes that all observations are independent and the estimation of the parameter of each arm's probability distribution is based on all observations in that arm:

$$r^k_1, \ldots, r^k_{n^k} \sim \mathrm{P}(\theta^k),$$

   where θ^k is the parameter for arm k.

2. In the no pooling situation, one assumes that all observations within users are dependent:

$$r^k_{1j}, \ldots, r^k_{n_j j} \sim \mathrm{P}(\theta^k_j).$$

3. In the partial pooling situation, we try to find a middle ground, and we here use a weighted combination of the above such that:

$$\tilde{\theta}^k_j = (1 - \beta_j)\theta^k_j + \beta_j \theta^k$$

$$r^k_{1j}, \ldots, r^k_{n_j j} \sim \mathrm{P}(\tilde{\theta}^k_j),$$

   where β_j is the aforementioned shrinkage factor.

One particularly simple choice for the shrinkage factor is $\beta_j = \frac{2}{2+n_j}$, where n_j is the number of observations for the user (Ippel et al., 2019). This approach is however informative to understand the desired behavior of a shrinkage factor: when the number of observations for a user n_j is zero, the shrinkage factor β_j is one, which then results in $\tilde{\theta}^k_j = \theta^k$. However, as n_j grows, $\tilde{\theta}^k_j$ approaches θ^k_j. Using this shrinkage factor is a simple and heuristic way of partial pooling.

There are, however, other shrinkage factors that are derived from and based on the assumptions of the data-generating model and its distributions. One example is the beta-binomial estimator. Using a method-of-moments estimation to estimate the shrinkage factor, we end up with the following shrinkage factor:

$$\beta^{BB}_j = \frac{\hat{M}}{\hat{M} + n_j}$$

where $\hat{M}$ is an estimator for $M = \alpha + \beta$ – see Ippel et al. (2019) for more details and full derivations. We will use both of these shrinkage factors in our simulation study.

3.3 Policies for the CMAB problem with dependent observations

Below we introduce three popular bandit policies, each with three different versions (complete pooling, no pooling and partial pooling), which we will use for the remainder of the study. We have chosen to use policies that are often used in the literature, which are ϵ-greedy (Sutton & Barto, 2011), Upper Confidence Bound (Auer et al., 2002; Li et al., 2011) and Thompson sampling (Agrawal & Goyal, 2012; Chapelle & Li, 2011), to make sure that we compare different policies of which we know what their expected behavior is. The partial pooling for ϵ-greedy and UCB will have two different versions, using either the heuristic shrinkage factor or the beta-binomial estimator.

3.3.1 ϵ-greedy

We first consider the ϵ-greedy (EG) policy. With EG, one randomly selects an action $a \in \mathcal{A}$ with probability ϵ, and with probability $1 - \epsilon$, one chooses the action with the highest expected reward. A general way of setting starting parameters would be $\hat{\theta}^a = 1/K$ for each $a \in \mathcal{A}$. This results in a policy that selects actions as follows:

Complete pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} \text{Rand}(a \in \mathcal{A}) & \text{if } u < \epsilon \\ \arg\max_a(\hat{\theta}^a) & \text{otherwise} \end{cases} \tag{3.2}$$

where u is a draw from $\mathrm{Uniform}(0, 1)$. For this policy (and UCB) the parameter $\hat{\theta}^a$ is a proportion which, each time a new reward comes in, is updated using

$$n^a_{t+1} = n^a_t + 1$$

$$\hat{\theta}^a_{t+1} = \hat{\theta}^a_t + \frac{r^a_t - \hat{\theta}^a_t}{n^a_{t+1}}$$

for each arm where n^a_t is the number of observations for that arm at time t.

No pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} \text{Rand}(a \in \mathcal{A}) & \text{if } u < \epsilon \\ \arg\max_a(\hat{\theta}^a_j) & \text{otherwise} \end{cases} \tag{3.3}$$

where $\hat{\theta}^a_j$ can be computed the same as for the complete pooling case, only with the rewards and number of observations for user j instead of all users.

Partial pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} \text{Rand}(a \in \mathcal{A}) & \text{if } u < \epsilon \\ \arg\max_a(\tilde{\theta}^a_j) & \text{otherwise} \end{cases} \tag{3.4}$$

where $\tilde{\theta}^a_j = (1 - \beta_j)\hat{\theta}^a_j + \beta_j\hat{\theta}^a$.

3.3.2 Upper Confidence Bound

Next, we consider the upper confidence bound (UCB) policy. This is an asymptotically optimal policy, first described by Lai and Robbins (1985), which works on the basis of optimism in the face of uncertainty. This means that, although the knowledge regarding the arms is lacking, the policy constructs an educated guess for the expected payoff of each arm. If the educated guess resulted in choosing an optimal action, then the policy is working optimally. If it is choosing an action (multiple times) that is not optimal, the policy will take learning of this and not choose the action in the future by adapting the educated guess.

Formally, UCB computes an upper bound for the confidence interval of the statistic of interest and it selects the action that maximizes the upper bound. There are multiple different types of confidence bounds considered in the literature (see e.g., Auer, 2002; Auer et al., 2002; Langford, 2005). In our case, we use one that is specifically derived from the beta-binomial distribution (Chapelle & Li, 2011; Langford, 2005). Formally the policy selects the arm with the highest

$$\hat{\theta}^a_t + \sqrt{\frac{2\hat{\theta}^a_t \log\frac{1}{\delta}}{n^a_t}} + \frac{2\log\frac{1}{\delta}}{n^a_t}, \delta = \sqrt{\frac{1}{t}},$$

where $\hat{\theta}^a_t$ and n^a_t are defined as before and $\sqrt{\frac{2\hat{\theta}^a_t \log\frac{1}{\delta}}{n_t}} + \frac{2\log\frac{1}{\delta}}{n_t}$ provides the upper confidence bound.

Before playing actions according to these bounds, the policy first plays all the arms once to receive a rough estimate of the statistic of interest. The actions that are not played often (i.e., have low n_t), will have a higher chance of being played later on. In the unfortunate event of not choosing the optimal arm, UCB eventually will be convinced to play it again due to the confidence bound. This results in the following definition of the policy and its different versions:

Complete pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} a & \text{if } n^a = 0 \\ \arg\max_a (\hat{\theta}^a + \sqrt{\frac{2\hat{\theta}^a \log\frac{1}{\delta}}{n^a}} + \frac{2\log\frac{1}{\delta}}{n^a}) & \text{otherwise} \end{cases} \tag{3.5}$$

No pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} a & \text{if } n^a_j = 0 \\ \arg\max_a (\hat{\theta}^a_j + \sqrt{\frac{2\hat{\theta}^a_j \log\frac{1}{\delta}}{n^a_j}} + \frac{2\log\frac{1}{\delta}}{n^a_j}) & \text{otherwise} \end{cases} \tag{3.6}$$

Partial pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \begin{cases} a & \text{if } n^a_j = 0 \\ \arg\max_a (\tilde{\theta}^a_j) & \text{otherwise} \end{cases} \tag{3.7}$$

where $\tilde{\theta}^a_j = (1-\beta_j)\Big(\hat{\theta}^a_j + \sqrt{\frac{2\hat{\theta}^a_j \log\frac{1}{\delta}}{n^a_j}} + \frac{2\log\frac{1}{\delta}}{n^a_j}\Big) + \beta_j\big(\hat{\theta}^a + \sqrt{\frac{2\hat{\theta}^a \log\frac{1}{\delta}}{n^a}} + \frac{2\log\frac{1}{\delta}}{n^a}\big)$.

3.3.3 Thompson sampling

Finally, Thompson sampling is a Bayesian policy in which an action a_t is randomly selected with a probability proportional to the belief that this action is the best action to play given some (Bayesian) model of the relationship between the action(s) and the rewards (Agrawal & Goyal, 2013b; Chapelle & Li, 2011). In its general form, one sets up a Bayesian model using some prior $P(\theta)$ to obtain posterior $P(\theta|\mathcal{D}) \propto P(\mathcal{D}|\theta)P(\theta)$. To subsequently select an action proportional to its probability of being optimal, it suffices to obtain a single draw θ' from the posterior $P(\theta|\mathcal{D})$ for each arm and then select the action with the largest posterior draw. In our specific case, using a (conjugate) $\mathrm{Beta}(\alpha, \beta)$ prior with the Bernoulli likelihood, the posterior becomes $P(\theta|\mathcal{D}) = \mathrm{Beta}(\alpha + R_c, \beta + n - R_c)$, with n the number of plays, R_c the reward, and hyperparameters α and β. We can specify a relatively non-informative (uniform) prior by setting $\alpha = 1$ and $\beta = 1$ as starting values. This results in the following policy (and versions thereof):

Complete pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \arg\max_a(\theta'^a) \tag{3.8}$$

where θ'^a is a single draw from the $\mathrm{Beta}(\alpha^a + R^a_c, \beta^a + n^a - R^a_c)$ posterior for arm a.

No pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \arg\max_a(\theta'^a_j) \tag{3.9}$$

where θ'^a_j is a single draw from the $\mathrm{Beta}(\alpha^a_j + R^a_{cj}, \beta^a_j + n^a_j - R^a_{cj})$ posterior for arm a and user j.

To implement partial pooling in a Bayesian setting, we use a Bayesian hierarchical model, which in essence results in setting hyperpriors on the hyperparameters α and β. Chapter 5 of Gelman et al. (2013) describes a way to specify a hyperprior for the beta distribution as follows: first, they define a reparameterization of α and β in terms of the mean $\phi = \frac{\alpha}{\alpha+\beta}$ of the beta distribution and $\kappa = \alpha + \beta$ where κ is roughly inversely related to the variance of the beta distribution. With some rearranging we get $\alpha = \kappa\phi$ and $\beta = \kappa(1 - \phi)$. Finally, we can set priors as follows: $\phi \sim \mathrm{Uniform}(0, 1)$ and $\kappa \sim \mathrm{Pareto}(1, 1.5) \propto \kappa^{-5/2}$. See Gelman et al. (2013) for more details on this hyperprior specification. This results in the following (simplified) policy:

Partial pooling:

$$\Pi(\mathcal{D}, x_t) \coloneqq a_t = \arg\max_a(\theta'^a_j) \tag{3.10}$$

where θ'^a_j is a single draw from the posterior $P(\theta^a_j|\mathcal{D})$, which is the resulting posterior given the hierarchical setup described above. In the complete and no pooling case posteriors are conjugate, which means that updating the posteriors is straightforward. In the hierarchical setting we need to resort to numerical approximations of the posterior. In the simulation study below we use Hamiltonian Monte Carlo (HMC) sampling (Carpenter et al., 2017). Note that we have only defined one partial pooling variant for Thompson sampling, and we do not use two different shrinkage factors as we do with UCB and ϵ-greedy. The partial pooling of Thompson sampling is comparable to the beta-binomial shrinkage factor used in UCB and ϵ-greedy, meaning that it is derived based on the assumed underlying reward distribution.

To conclude, we have provided three novel policies (the partial pooling versions of ϵ-greedy, UCB, and Thompson sampling for the Bernoulli bandit) to deal with possible hierarchical structures. Note that contrary to common CMAB approaches, our proposals do not specify an independent reward model for each unique value of the context (assuming J values of the context above, one for each user). In the next section we examine the performance of our proposals.

3.4 Simulation study

3.4.1 Design

In this simulation study, we compared the performance of the proposed adaptations and the standard solutions for the (C)MAB policies described above. The comparison was done in terms of expected cumulative regret (see Equation 4.1). To keep the study simple, the simulations were run using a 2-armed Bernoulli contextual bandit where the context was made out of a user identifier number j. We varied the following factors to examine the performance of the policies:

1. *Distribution of success probabilities:* Each unique user j had user specific success probabilities for each of the two arms that were drawn from a beta distribution (i.e., $\theta_j^a \sim \text{Beta}(\alpha, \beta)$). We used three different beta distributions to generate the true parameters. Firstly, we used $\text{Beta}(1.5, 1.5)$ for both arms as this has a high variance and ensures that for some users the first arm is optimal whereas for others the second arm is optimal. This approach ensures that the expected rewards for different arms will be different between users which is likely beneficial for policies that take hierarchical structures into account. Secondly, we used $\text{Beta}(5, 5)$ for both arms as this has a lower variance and the difference between arms for each user is smaller. Thirdly, we used $\text{Beta}(2.5, 1.5)$ (for arm A) and $\text{Beta}(1.5, 2.5)$ (for arm B) as a third distribution of success probabilities, which results in arm A being the better arm overall – but with enough overlap such that arm B is better only for some users.

2. *Number of users / repeated observations:* We varied the number of users $J \in \{50, 100, 500, 1000\}$. We generated $T = 10\,000$ interactions, which resulted in a different (average) number of observations per user $n_j \in \{200, 100, 20, 10\}$. A lower number of observations per user results in less information per user, and the policies that only focus on the observations within a user have a disadvantage. With a higher number of observations per user, exploiting the dependence between users is less beneficial, as there is enough information within the repeated measurements of the users.

3. *Distribution of user visits:* Sampling the context (i.e., user identifier) was done in two different ways. The first approach was to uniformly random sample a user identifier at each time t. As a result, all users approximately had the same number of observations $n_j \in \{200, 100, 20, 10\}$. This allowed us to investigate how a uniform increase in the number of observations per user affected the expected cumulative regret. Since in practice users rarely

have approximately equal numbers of observations we also used a second approach that is more in line with an empirically valid example (which was based on a data set we used in our empirical study): based on the average number of observations per user from that data set ($\lambda = 7.58$), we sampled the n_j's for each user from a Poisson distribution. Then, using the sampled n_j's, we created a normalized probability vector which we use to sample the user identifier also for $T = 10\,000$ interactions with the same variation in the number of users: this resulted in on average the same number of observations per user as in the uniform random sampling condition (i.e., $n_j \in \{200, 100, 20, 10\}$). Thus, while in the first case the number of observations for each user were relatively close together, in the second case there was quite a large variation between users.

In total, this means that there were 24 conditions in our simulation study (three levels for distribution of success probabilities, four levels for number of users and two levels for the distributions of user visits). Together, these 24 conditions were run for 1000 replications for each of the four (or three) different variations of the three policies (in total 11) generating a total of 264 simulations.

The simulation was implemented in the R package **contextual** (R Core Team, 2019; van Emden & Kaptein, 2018). As discussed before, for the partial pooling variant of Thompson sampling we resorted to HMC sampling, which was implemented in **Stan** (Carpenter et al., 2017). Since HMC sampling is computationally expensive, it was practically infeasible to recompute the posterior at each interaction. We chose to only sample from the posterior each 10 interactions. Furthermore, we used a warmup of 10 samples after which we took 10 draws and we re-used these draws in **Stan** each time such that **Stan** would start sampling the posterior from the region where it left off. The code for this paper can be found at https://github.com/Nth-iteration-labs/dependent-observations.

3.4.2 Results

Figure 3.3 shows the results for all the simulation conditions in terms of expected cumulative regret (the lower the regret, the better). We only show the results for the uniformly random sampling of the user identifier, since we found that the results for the uniform and Poisson distribution were almost identical. For clarity, only the average cumulative regret at the end of each simulation is shown. The results show a few general trends. Firstly, in almost all cases complete pooling performed worse than no pooling or partial pooling. This indicates that a naive bandit approach performs poorly. Secondly, with all policies (and their respective versions), the regret was higher when the number of users was higher (i.e., fewer

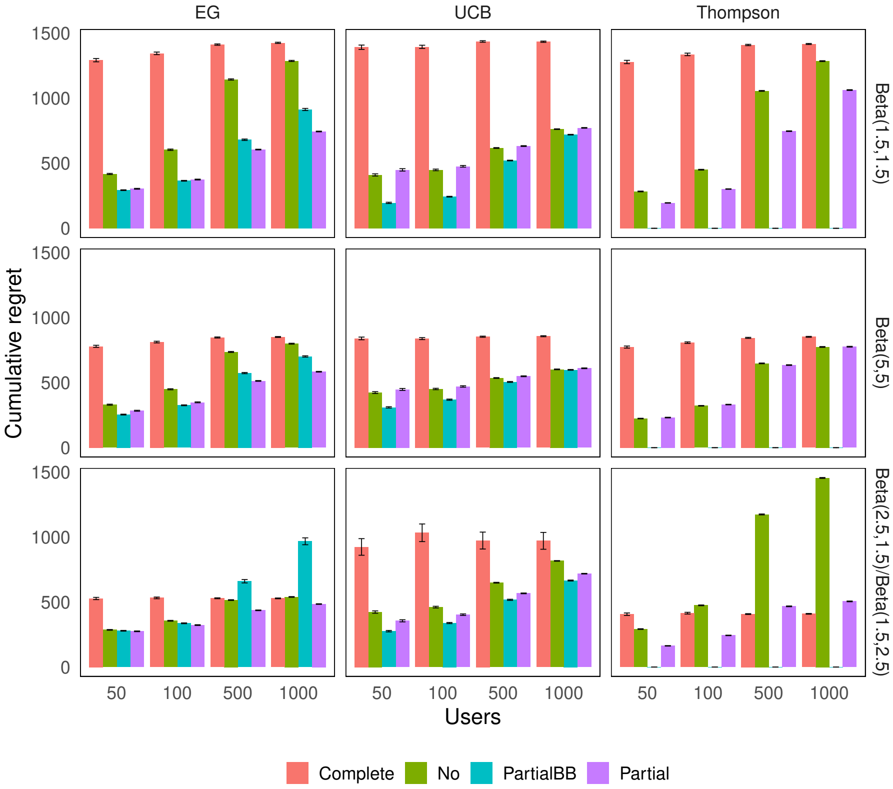


Figure 3.3: Results of the simulation study. Results for the Poisson condition are left out as they were almost identical to the uniform condition. *Partial BB* is partial pooling with the beta-binomial shrinkage factor and *Partial* with the heuristic shrinkage factor. *Partial BB* is not defined for Thompson sampling. Also shown are the error bars for the 95% confidence intervals. For all policies the partial pooling approach outperforms both the no pooling and the complete pooling approaches in most cases.

observations per user). Thirdly, comparing the two shrinkage factors with each other shows that they are very competitive to each other and have different performances depending on the scenario and policy (e.g., the beta-binomial shrinkage factor tends to perform better when used in conjunction with UCB). Finally, and as expected, in almost all cases the partial pooling approaches outperform both the no pooling and the complete pooling approaches: this indicates clearly that dealing with hierarchical dependencies in CMAB problems using a partial pooling approach is beneficial.

As for the performance of the policies, with ϵ-greedy we see that for the symmetrical beta distributions partial pooling works better, but the heuristic shrinkage factor performs better than the beta-binomial shrinkage factor. Also for the unsymmetrical beta distribution, we see that the heuristic shrinkage factor always performs better than the complete and no pooling variant. The beta-binomial shrinkage factor, however, suffers some regret increase for 500 and 1000 users.

With UCB we see that the beta-binomial shrinkage factor performed better compared to the heuristic shrinkage factor. The heuristic shrinkage factor performed roughly equivalent as the no pooling variant and thus showing not much improvement in the case of the symmetrical beta distributions. When it comes to the unsymmetrical beta distributions (the last row of the plot), we see that both shrinkage factors perform better than the complete and no pooling variants for UCB.

With Thompson sampling we see that for the symmetrical beta distributions (first two rows), the partial pooling variant works as good or better than the no pooling variant and always better than the complete pooling. With the unsymmetrical beta distribution, we see that no pooling greatly suffers a regret increase, but partial pooling not so much – albeit it performs slightly worse than complete pooling for 500 and 1000 users.

To summarize, we found that partial pooling performs in almost all cases better than their complete pooling and no pooling counterparts. This shows that incorporating hierarchical structures that might be present in the data (be it via partial pooling or otherwise) is of potential interest for future studies, as there is minimal loss in performance (in the worst case) and only a potential increase.

3.5 Empirical evaluation

In this study, we investigated the performance of the different pooling variations on an empirical data set previously reported upon in Kaptein et al. (2018) using offline evaluation (Li et al., 2011; Mary et al., 2014), which allows us to reuse in the field collected data to compare the performance of difference policies. In this experiment, users browsed products on a website. With each product page view, the users were shown at random a strategy that would try to persuade them to buy the product or no strategy at all (a control condition). These strategies were: authority (e.g., "recommended product"), social proof (e.g., "bestseller") and scarcity (e.g., "almost out of stock"). This data set is available in the provided repository.

3.5.1 Design

The data that we used consisted of the data where users browsed a product detail page, which displayed only a single product and a single strategy and adding the product to a shopping basket was counted as a success (i.e., the click-through

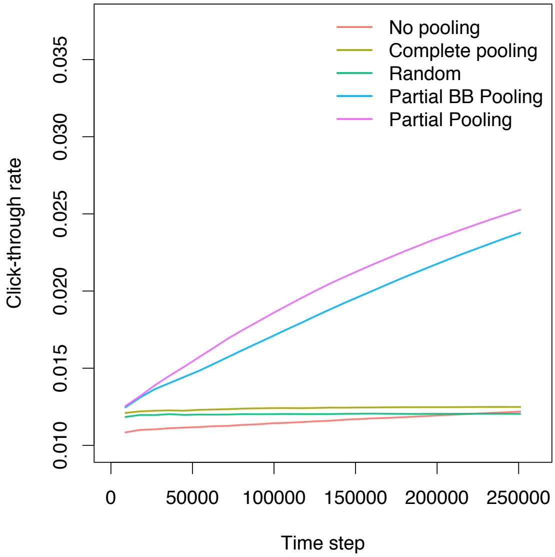


Figure 3.4: Results for the empirical evaluation for ϵ-greedy. The plot shows the click-through rates (CTR) over time for each pooling type for ϵ-greedy. We can see that the partial pooling variants heavily improve the CTR.

rate, CTR). The data consisted of $570\,061$ observations with an average of 7.58 observations per user (with a total of $75\,132$ users). We have selected the users that had a minimum of 5 (to ensure enough information) and a maximum of 50 (to ensure that no bots would be included) observations. This left us with $23\,436$ users with $327\,600$ observations.

For the policies, we only considered ϵ-greedy and UCB policies and not Thompson sampling, as these policies have shown to have both an improvement in terms of regret and are computationally feasible to use. Furthermore, next to the four variants of the policies, we have added a random policy that selects actions uniform randomly to act as a performance baseline. The data can be considered as containing four arms – the three different persuasion strategies and also a control group where no strategy was displayed.

3.5.2 Results

Figure 3.4 and Figure 3.5 show the results for the empirical study in terms of the CTR. For both UCB and ϵ-greedy, the partial pooling variants seemed to improve the results compared to their baselines (with UCB complete pooling and the random policy overlap in the plot). With ϵ-greedy we see again that the heuristic shrinkage factor performs better. With UCB we can see that in the beginning the

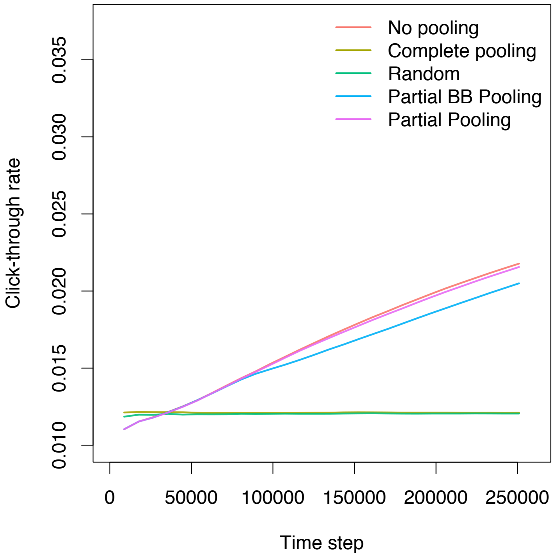


Figure 3.5: Results for the empirical evaluation for UCB. The plot shows the click-through rates (CTR) over time for each pooling type for UCB. We see that the partial pooling at least improves the complete pooling, but performs as well as the no pooling variant.

beta-binomial partial pooling variant follows the no pooling variant, resulting in a lower CTR in the beginning, but both the no pooling and the heuristic shrinkage factor will perform better after a certain point. The heuristic shrinkage factor has again (as in the simulation) the same performance as the no-pooling variant for UCB – these lines almost completely overlap. Overall, this offline evaluation using real-world data provides an externally valid demonstration that partial pooling is beneficial.

3.6 Conclusions

In this paper we introduced partial pooling as a means of dealing with dependent observations in the (contextual) multi-armed bandit problem. Both our simulation and empirical study showed that adapting policies to exploit hierarchical structures in the data improves the performance of these policies. Taking hierarchical structures into account can be beneficial for the CMAB problem and does not exclusively pertain to the situations shown in this paper: based on the overwhelming utility of hierarchical modeling in the social sciences we expect similar results for different (i.e., non-Bernoulli) reward distributions or when multiple (cross-)nestings are present in the data.

Future research can be done to build upon the work shown in the paper. To improve UCB further we could in the future derive shrinkage factors that are optimal under specific assumptions regarding the environment: thus, our focus on hierarchical structures opens up a treasure trove of new theoretical challenges. We tried to show, however, through an empirical approach that our general approach is useful. For Thompson sampling the partial pooling worked well in terms of improving rewards, but it comes at high computational cost. Further research could be done on improving the computational cost, for example by using sequential Monte Carlo methods (Doucet, De Freitas, & Gordon, 2001) or by implementing bootstrap Thompson sampling (Eckles & Kaptein, 2019).

Despite the fact that we focussed on a relatively simple example in terms of the imposed hierarchical structure, we would like to stress that hierarchical structures are very common in practice: in online marketing for example we find user, page and topic hierarchies (Ansari & Mela, 2003; Cheng & Cantú-Paz, 2010). To our surprise, we find almost no research in the bandit literature that takes these hierarchical structures explicitly into account, despite the fact that in science hierarchical models are common-place. We hope to have demonstrated that hierarchical models – and the associated partial pooling of information within and between clusters – can be of large benefit for CMAB problems and deserve more research attention.

Chapter 4

An Empirical Comparison of Offline Evaluation Methods for the Continuous-Armed Bandit Problem

Abstract

The (contextual) multi-armed bandit (MAB) problem provides a formalization of sequential decision-making which has many applications. However, validly evaluating MAB policies is challenging; we either resort to simulations which inherently include debatable assumptions, or we resort to expensive field trials. Recently several *offline* evaluation methods have been suggested that are based on empirical data, thus relaxing some of the assumptions, and can be used to evaluate multiple competing policies in parallel. These methods are however not directly suited for the continuous-armed (CAB) problem; an often encountered version of the MAB problem in which the action set is continuous instead of discrete. We propose and evaluate a novel offline evaluation method developed specifically for the evaluation of CAB policies. We empirically demonstrate that our method provides a relatively consistent ranking of policies. Also, we compare our approach to recent alternatives in a simulation study for parameter tuning. Finally, we detail how our method can be used to select policies in a real-life CAB problem.

Keywords: offline evaluation, continuous-armed bandit problem, continuous treatments, dynamic policies

Submitted as Kruijswijk, J.M.A., Parvinen, P., Kaptein, M.C. An Empirical Compirason of Offline Evaluation Methods for the Continuous-Armed Bandit Problem.

4.1 Introduction

In the canonical multi-armed bandit (MAB) problem a gambler stands in front of a row of slot machines, each with a (potentially) different payoff. It is up to the gambler to decide in sequence which machine to play and, during the course of sequentially playing the machines, she aims to make as much profit as possible by simultaneously learning from the previous observations and using the gained knowledge to steer future actions (Berry & Fristedt, 1985; Whittle, 1980). The gambler needs to pick a *strategy* that dictates which arm to play next given the previous observations.

The problem of finding such a strategy is complicated since at each interaction the gambler only observes the outcomes of the machine she played, and she will never know the outcomes of the other possible courses of action at that moment in time. This so-called omission of *counterfactuals* (Li et al., 2011) – not being able to gain knowledge about all the possible outcomes – gives rise to the exploration versus exploitation trade-off (Berry & Fristedt, 1985): at each time point an action can either be geared at gaining more knowledge regarding the machines she is uncertain about (exploration), or it can be geared at using the knowledge gained in earlier interactions by playing machines with a high expected pay-off (exploitation). A good strategy balances this trade-off and does not waste too many plays on gaining new knowledge, nor does it become too greedy and get stuck exploiting a suboptimal machine (Kaelbling et al., 1996).

The MAB problem is easily extended to more general settings. One such extension is the contextual MAB (CMAB) problem (Langford & Zhang, 2008). In the CMAB problem, at each interaction, the gambler observes the state of the world (context), which might influence the optimal choice at that moment in time (Beygelzimer et al., 2011; Bubeck & Cesa-Bianchi, 2012; Langford & Zhang, 2008). Both the MAB problem and the CMAB problem have been heavily analyzed (Wang et al., 2005). Furthermore, strategies, or in this literature more often called *policies*, to address the (C)MAB problem have found many practical applications in recent years: examples include, but are not limited to, the personalization of online news (Li et al., 2010), online advertisement selection (Cheng & Cantú-Paz, 2010), website morphing (Hauser et al., 2009), adaptive clinical trials (Press, 2009; Williamson et al., 2017), and software to experiment with bandit policies on the web (Kruijswijk, van Emden, Parvinen, & Kaptein, 2020).

In the current article we focus on another extension of the MAB formalization coined the continuous-armed bandit (CAB) problem (Agrawal, 1995): this problem distinguishes itself apart from other formalizations by considering instead of a set of discrete actions (the distinct slot machines) a continuous range of actions. The CAB problem has also been analyzed (Agrawal, 1995; Kleinberg, 2004; Krause & Ong, 2011), however the current theory focussed literature lacks applied methods to evaluate the performance of different CAB policies. This is true despite many applied settings in which this problem is encountered. Which include but are not limited to, choosing an optimal price for selling a product to customers encountered sequentially (Javanmard & Nazerzadeh, 2019), or choosing an optimal treatment dose (Kallus & Zhou, 2018). In this paper we suggest and evaluate a practical method for evaluating the performance of different CAB strategies in an externally valid setting. Notably, we compare the performance of our method with a recently published alternative.

As stated above, the omission of counterfactuals complicates finding a good policy since it gives rise to the exploration-exploitation trade-off. Counterfactuals also complicate the evaluation of competing policies: due to the omission of counterfactuals in the collected data resulting from field evaluations of a specific policy, it is challenging to use these existing data to directly evaluate alternative strategies. Therefore, if we want to empirically evaluate bandit policies we either have to resort to running multiple field evaluations that are often very expensive to carry out, or we have to resort to simulation based methods which often lack external validity. Li et al. (2011) suggested an effective solution to this problem for the (C)MAB problem: they proposed a method for the externally valid *offline* – thus based on existing, pre-collected, data – evaluation of MAB policies. The method relies on a single dataset, thus cutting costs, while it circumvents the validity problems that easily arise in simulations by using actual empirical data.

The offline MAB evaluation method suggested by Li et al. (2011) relies on collecting – in the field – a dataset in which the actions where taken uniformly at random at each interaction. Next, to evaluate a particular decision policy, the sequence of data points is replayed and, at each interaction, the action suggested by the policy under evaluation is compared to the action that is actually present in the logged data at that point in the sequence. If the two actions match, the data point gets “accepted” and its outcome is included in the evaluation of the policy. If the actions do not match the interaction is simply ignored. This method demonstrably provides unbiased estimates of the performance of distinct bandit policies (albeit for a smaller number of interactions than the number of datapoints collected in the initial field trial).

In practice, the method by Li et al. (2011) works well only when the number of actions is (relatively) low, the amount of observations is large, or both. If however the action set is large, or the number of observations in the dataset small, estimates of the performance of the different policies can only be obtained for a (very) small number of interactions. Cleary, in the limit, the suggested method thus fails for the CAB problem; since theoretically the number of possible actions is infinite, the probability that the actions suggested by a policy under evaluation matches the randomly selected action in the existing dataset tends to zero. As a result, no observations will be accepted and the evaluation of the policy fails.

Other methods for offline evaluation exist such as the method introduced by Dudík, Langford, and Li (2011). The advantage of this method is that the dataset does not have to be collected uniformly randomly, as long as the probability of playing the action that was played is known (the propensity score). Some research has been done on computing propensity scores when continuous treatments are considered, but it still remains a complicated problem (Hirano & Imbens, 2004). In practice, the method by Li et al. (2011) is a special case of the method by Dudík, Langford, and Li (2011): when choosing actions uniform randomly the propensity score is a constant and therefore it can be ignored. This means that the method by Dudík, Langford, and Li (2011) shares a number of the same drawbacks for offline evaluating policies for the CAB problem.

Kallus and Zhou (2018) recently introduced an offline evaluation method for policies which consider continuous treatments. More specifically, the method is designed to evaluate policies that we consider *static*. Static policies are policies that have their parameters completely defined a priori. However, we are also interested in offline evaluating *dynamic* policies: contrary to static policies, dynamic policies are treatment allocation policies that change their behavior based on the data collected (and thus do not have their parameters defined a priori). For example, we consider online linear regression in combination with ϵ-greedy a dynamic policy, as the coefficients of the model are updated during the interactions. Dynamic policies are contrasted to static policies in which effectively the mapping from context to action at each timepoint is known a priori (i.e. a model with the coefficients already known). Focusing on dynamic policies makes offline evaluation even harder as now we need to not only focus on the overall outcomes, but we are also interested in the learn rates during the sequential allocation procedure.

In this paper, we present and empirically evaluate a logical extension to the method of Li et al. (2011) to make it suitable for the evaluation of CAB policies. We evaluate the method in a case where the aim is to rank a number of dynamic policies based on their expected performance. Furthermore, we compare our method to the method of Kallus and Zhou (2018) in a use case where the aim is to tune and choose the learning parameters of a policy for the CAB problem. In the next section, we introduce the CAB problem more formally and provide examples of CAB policies. Next, we introduce the method by Li et al. (2011) and our extension to the method and we discuss its rationale. We also give an introduction to the method by Kallus and Zhou (2018). Then we evaluate the performance of our suggested method by showing through simulations that it allows one to consistently order bandit policies for CAB problems, for multiple sizes of the problem and for multiple true data generating models. Then we will showcase how parameter tuning would work in an *online* scenario (i.e. we have multiple experiments to try different parameters). After this we will compare the two methods to do offline parameter tuning and compare how they hold up against the online case. Finally, we present the use of our method in the field of online marketing and discuss future research directions and possible improvements of our method. While our suggested method does not provide unbiased estimates of the absolute performance of the evaluated bandit policies – we explicate this below – it does provide a cheap and straightforward method to provide a relative rank of distinct policies and thus aid decision making when selecting policies for applied CAB problems – even when the policies concerned are dynamic, i.e. changing over time.

4.2 Continuous-armed bandit problem

Before we introduce our extension to the method proposed by Li et al. (2011), we first more formally introduce the MAB (and CAB) problem. Bandit problems can be described as follows: at each time $t = 1, ..., T$, we have a set of possible actions $\mathcal{A}$. After choosing $a_t \in \mathcal{A}$ we observe reward r_t. The aim is to find a *policy* ($\Pi(h_{t-1})$ where h_{t-1} is the historical data), which is a mapping from all the historical data to the action at t, to select actions such that the cumulative reward $R_c = \sum_{t=1}^{T} r_t$ is as large as possible. In the case of a CAB problem, the same formalization can be used, where the only difference is in the action set: in the CAB problem we have $\mathcal{A} \in \mathbb{R}$ (often constrained within some range $[i, j]$).

To assess how a policy performs we often look at the expected regret of the policy which is defined by

$$\mathbb{E}[R_T] = \mathbb{E}\left[\sum_{t=1}^{T} r_t^* - r_t\right] \tag{4.1}$$

where r_t is the reward at interaction t and r_t^* is the reward of the action with the highest expected reward (the optimal policy). Regret, as opposed to the cumulative reward, R_c, provides an intuitive benchmark since a perfect strategy would incur an expected regret of 0. Further note that if a suboptimal action has a nonzero and non-decreasing probability of being selected, the regret will – in expectation – increase linearly. Most analytical work focusses on showing the asymptotic sub-linear regret of distinct (C)MAB or CAB policies (e.g., Lai & Robbins, 1985).

4.2.1 Continuous-armed bandit policies

Following are a few illustrations of CAB policies. We use these policies in the simulation studies to evaluate our proposed method. The policies considered in this paper are:

1. The *Uniform random, UR,* policy. In this policy simply $a_{1,\ldots,T} \sim$ Unif(i,j). The regret of this policy is expected to grow linearly.

2. The *ϵ-first, EF,* policy. This is a greedy algorithm that has two phases. In the first phase, which is restricted by a preset N number of interactions, the policy *explores*: $a_{1,\ldots,N} \sim$ Unif(i,j). Next, a simple linear model is fit to the observed data. The model that we fit using standard least squares estimation is

$$r = \beta_0 + a\hat{\beta}_1 + a^2\hat{\beta}_2 \tag{4.2}$$

   and subsequently, in the exploit stage we choose the action that maximizes this fitted curve, $a_{N+1,\ldots,T} = \frac{-\hat{\beta}_1}{2\hat{\beta}_2}$. The regret of this policy is expected to grow linearly in both phases, however, in the first phase it will grow faster than in the second phase, since it uses the expected (heuristically) optimal action in the second phase (and stops exploring).

3. The *Thompson sampling using Bayesian linear regression, TBL,* policy. Thompson sampling (Agrawal & Goyal, 2012; Scott, 2010; Thompson, 1933) is a sampling method in which an action a_t is randomly selected with a probability proportional to the belief that this action is the best action to play given some (Bayesian) model of the relationship between the actions and the rewards. Consider all the historical data, previously denoted h_t, consisting of the history of the actions and rewards up to t. Further denote the parameters $\theta = \{\beta_0, \beta_1, \beta_2\}$ (as in Equation 4.2). We set up a Bayesian model using some prior on $P(\theta)$ and obtain posterior $P(\theta|h_t) \propto P(h_t|\theta)P(\theta)$. To subsequently select an action proportional to its probability of being optimal – and thus to implement Thompson sampling – it suffices to obtain a single draw θ'_t from the posterior $P(\theta|h_t)$ and then select the action that is optimal given the current draw using $a_t = \frac{-\hat{\beta}'_1}{2\hat{\beta}'_2}$ (Scott, 2010). To compute $P(\theta|h_t)$ we use – using matrix notation – the well known online Bayesian linear regression model (as described, e.g., by Box & Tiao, 1992) where we update at each

time point:

$$J := J + \frac{r_t \boldsymbol{a}^T}{\sigma^2}, \tag{4.3}$$

$$P := P + \frac{\boldsymbol{a}\boldsymbol{a}^T}{\sigma^2} \tag{4.4}$$

and where $J = \Sigma^{-1}\mu$ (i.e. the precision times the mean), $P = \Sigma^{-1}$ (i.e. the precision) and $\boldsymbol{a} = [1, a, a^2]$. Finally, we sample the sought after $\theta | h_t \sim \mathcal{N}(\mu, \Sigma)$ from which draws are obtained at each time point. We again use the model presented in Equation 4.2. The regret of this policy is expected to grow sub-linearly.

4. The *Lock-in Feedback, LiF,* policy. LiF is a novel algorithm developed by Kaptein and Ianuzzi (2016) (see also Kaptein, van Emden, & Iannuzzi, 2016b). LiF is inspired by a method that is frequently used in physics, coined lock in amplification (Scofield, 1994; Wolfson, 1991) that is routinely used to find – and lock in to – optima of noisy signals. LiF works by oscillating sampled values with a known frequency and amplitude around an initial value a_0. Using the observed feedback from the oscillations in the evaluations of $f()$ it is straightforward to find the derivative $f'()$ at a_0 and use a gradient ascent updating scheme to find a_t^*, see Algorithm 1 for details. This function is expected to grow linearly in terms of regret, although it is expected that it grows slower than the first two policies (i.e., it reaches an optimum faster, but because of its oscillating nature the expected regret keeps growing linearly).

4

Algorithm 1 Implementation of the LiF policy as used in our evaluations. Here T denotes the total length of the data stream of "accepted" actions.

```
Inputs: value $a_0$, amplitude $A$, integration window $i$, learn rate $\gamma$, and frequency $\omega$
$r_\omega^\Sigma \leftarrow 0$ (cumulative rewards)
for $t = 1, \ldots, T$ do
    $a_t = a_0 + A \cos \omega t$
    $r_t = f(a_0 + A \cos \omega t) + \epsilon_t$
    $r_\omega^\Sigma = r_\omega^\Sigma + r_t \cos \omega t$
    if $(t \mod i == 0)$ then
        $r_\omega^* = r_\omega^\Sigma / i$
        $a_0 = a_0 + \gamma r_\omega^*$
        $r_\omega^\Sigma = 0$
    end if
end for
```

The above policies where chosen to a) include a very naive benchmark (the UR policy), and a number of different approaches advocated in the (c)MAB or CAB literature (Box & Tiao, 1992; Kaptein & Ianuzzi, 2016; Sutton & Barto, 2011). Please note that the number of possible alternative policies we could have explored is extremely large, ranging from simple heuristic strategies such as ϵ-greedy (Sutton & Barto, 2011) to currently popular Gaussian processes (Djolonga, Krause, & Cevher, 2013); we hope however to have included a selection of policies that provides an informative evaluation of the merits of our proposed method.

4.3 Offline CAB policy evaluation

In many applied situations we have no knowledge about the actions with the highest expected reward (i.e. we do not know r^*) and thus we will not be able to compute the regret. In such cases the best we can do is compare the cumulative reward R_c (or the average per time point reward R_c/T) obtained over multiple comparable runs – either in simulations or in field evaluations – of the policies under evaluation. However, this highlights a clear challenge when evaluating multiple policies: simulations likely contain assumptions that limit the external validity of the evaluation, while in-field evaluations of multiple policies are often difficult and expensive to carry out.

To address these problems Li et al. (2011) proposed a method to obtain unbiased estimates of the expected cumulative reward of different policies using a single, externally valid, dataset. Algorithm 2 details the proposed method: we run sequentially through a stream of logged data in which the actions have been selected uniformly at random. At each event in the stream, the policy under evaluation proposes an action. If the action proposed by the policy is the same as the action of the logged event, then the event is counted towards the evaluation of the policy and the observed reward is added to the total payoff. Note that if there are K actions, then the number of valid events T in the evaluation process is a random number with expected value L/K, where L is the length of the logged data set. Thus, during the evaluation datapoints are “accepted” with probability $p_{accept} = \frac{1}{K}$. In the online setting, $L = T$ and in the offline setting $\mathbb{E}(T) = \frac{L}{K}$. Here we consider the online setting our true evaluation of the policies (i.e. based on simulations that do not need offline evaluation).

Li et al. (2011) show that, under a number of assumptions regarding the collection of the logged dataset and the stationarity of the process, the method described in Algorithm 2 provides an unbiased estimate of the performance of policy Π. As such, the method makes it possible to compare multiple competing policies in an externally valid setting without the recurring costs of repeating field trials. In practice, however, the method fails for the often encountered CAB problem. This is due to the fact that with continuous action space the probability that a logged action is equal to a suggested action by the policy is very low: as K grows p_{accept} decreases and we have for the CAB problem $K \rightarrow \infty$ and $p_{accept} \rightarrow 0$. Thus, the method fails.

Algorithm 2 Policy evaluator with finite data stream for the MAB problem.

```
Inputs: policy $\Pi$; stream of events $S$ of length $L$
$h_0 \leftarrow$ (An initially empty history)
$R_c \leftarrow 0$ (An initially zero total payoff)
$T \leftarrow 0$ (An initially zero counter of valid events)
for $t = 1, 2, \ldots, L$ do
    Get the $t$-th event $(a, r_a)$ from $S$
    if $\Pi(h_{t-1}) = a$ then
        update $\Pi(r_a, a)$
        $h_t \leftarrow$ CONCATENATE$(h_{t-1}, (a, r_a))$
        $R_c \leftarrow R_c + r_a$
        $T \leftarrow T + 1$
    else
        $h_t \leftarrow h_{t-1}$
    end if
end for
Output: $R_c$ and $T$ (or $R_c/T$ for the average reward)
```

4.3.1 The delta method

In an attempt to solve this problem and to provide a practically usable method for the offline evaluation of CAB policies we propose an alternative to the method suggested by Li et al. (2011), which we call the delta method. Algorithm 3 describes our logical adaptation of Algorithm 2 to provide an evaluation method for the CAB problem. The difference between the two algorithms is in the **if** statement that determines acceptance of the proposed action: instead of constraining the suggested action to be *exactly* equal to the logged action, we compare the distance between the action logged in the dataset, a, with the action proposed by the policy, $\Pi(h_{t-1})$. If the absolute distance between these two actions is less than the tuning parameter δ, we accept the data point, and else it will be discarded. Intuitively the proposed change corresponds to the difference between the evaluation of a PDF of a discrete versus a continuous random variable. Note here that we update the policy not with the logged action a but rather with the action that was proposed by the policy ($\Pi(h_{t-1})$). Hence providing a noisy estimate of the reward at $\Pi(h_{t-1})$.

Algorithm 3 Offline policy evaluation for the CAB problem

```
Inputs: policy $\Pi$; stream of events $S$ of length $L$ with actions selected randomly in the range $[i, j]$
$h_0 \leftarrow$ (An initially empty history)
$R_c \leftarrow 0$ (An initially zero total payoff)
$T \leftarrow 0$ (An initially zero counter of valid events)
for $t = 1, 2, \dots, L$ do
  Get the $t$-th event $(a, r_a)$ from $S$
  if $|a - \Pi(h_{t-1},)| < \delta$ then
    update $\Pi(r_a, \Pi(h_{t-1}))$
    $h_t \leftarrow$ CONCATENATE$(h_{t-1}, (\Pi(h_{t-1}), r_a))$
    $R_c \leftarrow R_c + r_a$
    $T \leftarrow T + 1$
  else
    $h_t \leftarrow h_{t-1}$
  end if
end for
Output: $R_c$ and $T$
```

4.3.1.1 Properties of the delta method

Before we empirically evaluate the applied use of proposed method in an extensive simulation study in the next sections, it is worthwhile to analyze the role of the tuning parameter δ and to reflect on the resulting estimates of R_c that follow from our procedure. This is most easily done by keeping in mind a very simple CAB formalization where the true data generating process is merely a parabola constrained within the range $[0, 1]$, say $r_t = f(a_t) = -(a_t - .5)^2 + \epsilon$ where ϵ represents some random noise and we have $\mathbb{E}(\epsilon) = 0$. Note that $\Pi^*(t) = a_t^* = .5$. Clearly, a large value of δ (e.g., $.25$) will lead to accepting a high number of proposed actions (and thus large number of evaluations T), but will also lead to high variation in the realizations of $f(a_t)$: the policy evaluates $f()$ at $\Pi(h_{t-1}, x)$, and receives as a result $f(a_t)$ which might be at most δ away. Hence, for large δ, the performance of the policy will be poor since it obtains erroneous evaluations of $f(a_t)$, and the estimated cumulative regret will be (severely) biased. The exact way in which a policy will be biased by these erroneous evaluations of $f()$ heavily depends on the way in which the policy incorporates the history h_{t-1} when selecting the next action. These problems diminish as δ decreases, however, the number of accepted observations T will decrease accordingly. Hence, δ should be chosen as small as possible, however the expected number of accepted events $T = p_{accept}L = \frac{2\delta}{b-a}L$ (with range of actions $[i, j]$) should be as close as possible to the expected number of events occurred in the real-life setting for which the policy is evaluated. This implies that in practice one would like to collect a dataset containing uniformly randomly selected actions in range $[i, j]$ with length $L = \frac{2\delta}{(a-b)/T'}$ where T' is the desired length of the policy evaluation for the applied problem.

Note that as long as $\delta > 0$ the estimated expected reward $\mathbb{E}(R_c)$ is downwardly biased for concave functions since even if the policy converges exactly on the optimal action (e.g., selecting $a_{t=t',\dots,t=T} = a^* = .5$ for some t') the evaluations of $f(a_t)$ originate uniformly randomly from the interval $[a^* - \delta, a^* + \delta]$. Since each evaluation for which $a_t \neq a^*$ leads, in expectation, to a reward $r_t \leq r_t^*$, the expected cumulative reward of the policy under consideration is downwardly biased. Nonetheless, for the comparison of the *relative* performance of applied CAB policies this is not as cumbersome as it might sound; as long as the ranking of policies is relatively consistent for the desired scale of the problem T the method is still useful to select one out of a number of competing policies. We will demonstrate below that this is the case for a range of values of δ as well as for multiple true data generating functions $f()$. Hence, our proposed method is valuable in practice.

4.3.2 The kernel method

In Kallus and Zhou, 2018 the authors have developed an offline evaluation method for policies with continuous treatments that results in a consistent estimator in terms of bias and variance. It can use both the doubly robust (see Dudík et al., 2014) and the inverse propensity score weighting (IPW) methods (see Horvitz and Thompson, 1952). IPW methods are used when the original actions in the logged data set are not generated uniform randomly. The IPW is then used to weight the reward based on the probability of being chosen (the propensity score). Both these cases we do not consider and we assume that the propensity score (later named Q) is equal for every action. In Kallus and Zhou, 2018 the authors propose an alternative for the IPW method by applying a smoothing (kernel) function on the if-statement in Algorithm 2, which results in an algorithm as shown in Algorithm 4 (hereafter called the kernel method). Here, H is the horizon (a tuning parameter), $\boldsymbol{K}$ a kernel function (either a Gaussian or a Epanechnikov kernel) and Q the propensity score. The if-statement in Algorithm 2 is used to compare the suggested action with the logged action in the dataset. The kernel smoothing is made to deal with this problem by weighting the rewards: the further away an action, the reward is included less and less.[1]

The downside of this evaluation method is that in practice every selected action will be accepted and evaluated. This means that when we select an action a that is far away from the logged event, the policy will receive a reward near 0. Consider the ϵ-first, LiF and TBL policies: they have learning parameters that determine how they update the policy itself based on the data collected. Since

[1] Note that this only works for relatively smooth functions.

Algorithm 4 The kernel method

```
Inputs: policy $\Pi$; stream of events $S$ of length $L$
$h_0 \leftarrow$ (An initially empty history)
$R_c \leftarrow 0$ (An initially zero total payoff)
for $t = 1, 2, \ldots, L$ do
    Get the $t$-th event $(a, r_a, Q_a)$ from $S$
    Set evaluated reward $r_e = \boldsymbol{K}\left(\frac{\Pi(h_{t-1})-a}{H}\right)\frac{r_a}{Q_a}$
    update $\Pi\,(r_e, \Pi(h_{t-1})$
    $h_t \leftarrow$ CONCATENATE$(h_{t-1}, (\Pi(h_{t-1}), r_e))$
    $R_c \leftarrow R_c + r_e$
end for
Output: $R_c/(LH)$
```

the actual rewards are not observed, but rather a downweighted version thereof, the weights will need to make their way into the update of the parameters of the policy; this is often not straightforward.

We consider policies that have learning parameters as *dynamic* policies: the policies will change their behavior (e.g. update parameters) based on the data that are collected. They are contrasted to *static* policies, which do not update any parameters (and are not constrained by learning parameters) over time. Effectively this means that any policy that uses historical data (i.e. data collected during the "run" itself) is dynamic. Dynamic policies are negatively affected by the kernel method, as they will receive heavily biased rewards and update according to those rewards, possibly resulting in wrongly updated policies. Nevertheless, we will consider the kernel method also in a simulation study where we look at tuning the learning parameters of two policies and compare the kernel method with our delta method.

4.4 Simulation study

To evaluate our proposed method we perform two simulation studies. We have implemented these simulations in the R package **contextual** (van Emden & Kaptein, 2018). **Contextual** allows us to quickly implement different bandit policies and evaluate them for multiple iterations.[2] We first examine the performance of the four introduced CAB policies while collecting data "online" – as we know the true data generating model, we are able to evaluate the proposed actions by the policy directly into the models and observe their rewards and run as many evaluations as we want. Next, we collect an offline evaluation dataset S by gathering rewards $r_{1,\ldots,L}$ from our data generating models for actions chosen uniformly

[2] The implementation can be found on https://github.com/Nth-iteration-labs/offline-evaluation-continuous-bandits.

at random: $a_{1\ldots,L} \sim$ Unif$[0,1]$. Subsequently, to evaluate the offline evaluation methods, we can compare the performance of the policies between the online and offline scenarios.

We specify two different true reward functions, which we will use for both simulation studies. We have data generating models $f_1()$ and $f_2()$ of which we want to find the optimum. In a field experiment we would not know these models, but here we assume it takes one of the following two forms:

$$r_t = f_1(a_t) = -(a_t - c)^2 + 1 + \epsilon \tag{4.5}$$

with $c \sim \mathcal{U}(0.25, 0.75)$ and $\epsilon \sim \mathcal{N}(0, 0.01)$ and

$$r_t = f_2(a_t) = g(x; \mu_1, \sigma_1, a, b) + g(x; \mu_2, \sigma_2, a, b) + \epsilon \tag{4.6}$$

where $g()$ is the density function of the truncated normal distribution and $\mu_1 \sim \mathcal{U}(0.15, 0.2)$, $\mu_2 \sim \mathcal{U}(0.7, 0.85)$, $\sigma_1 \sim \mathcal{U}(0.1, 0.15)$, $\sigma_2 \sim \mathcal{U}(0.1, 0.15)$, $a = 0$, $b = 1$ and again $\epsilon \sim \mathcal{N}(0, 0.01)$. In practice, Equation 4.5 is a unimodal and Equation 4.6 is a bimodal function (and hence more challenging for any CAB policy). The two reward functions used were both varied across simulation runs (i.e. the means and variances vary) to not favor particularities of the distinct policies under scrutiny.

We conducted both simulation studies using the data generating processes described above. The first study focusses on ranking policies in terms of regret and comparing their online (or true) behavior versus their behavior using the delta method. The second study focusses on choosing the learning parameters that will result in the highest cumulative reward for a given number of interactions using both the delta and kernel offline evaluation methods.

4.4.1 Policy ranking

In this simulation study we first ran 10^3 repetitions of $L = T = 10^4$ interactions to obtain online estimates of the performance of the 4 policies; this provides our benchmark. Next, we ran 10^3 repetitions, each on a data set S of length $L = 10^4$ with different values of $\delta \in (.01, .05, .1, .2, .5)$, of Algorithm 3 for each of the 4 policies under consideration. This results in multiple offline evaluations of the same policies which we subsequently compare with the true online performance. Note that the total length of the offline evaluations differs for different values of δ. Also note that the reward functions $f()$ also differed for the online and offline situations, each repetition we randomly generated a different $f()$ (as explained above). In the simulation studies we have chosen for a smaller practical scale in terms of number of interactions, as a simulation study on a larger scale might be interesting to investigate the properties for an infinite horizon for T, but a smaller scale is more useful for situations where we know that the amount of data will be rather small. For LiF, we chose starting values $a_0 \sim \mathcal{U}(0, 1)$, $A = 0.05$, $i = 25$, $\gamma = .1$, $\omega = 1$. And for TBL this was $J = [0, 0.25, -0.25]$ and $P = \text{diag}(2, 2, 5)$.

In this simulation study, we will only look at the delta method. The main reason for this is that using the kernel method for evaluating dynamic policies is not trivial. While the kernel method provides accurate estimates of the (average) reward for static policies, the inherent weighting of observations makes it challenging to implement and evaluate dynamic policies validly. For any policy in which the behavior is changed due to the incoming data stream, the weights have to be accounted for thus making a change to the original policy. While in many cases this might be possible, one would still not be evaluating the actual policy but rather a revised version of the policy especially tailored to conform to the specific kernel being used.

4.4.1.1 Results

The unimodal model Figure 4.1 shows the results for the unimodal model in terms of empirical regret (see Equation 4.1). The upper left panel shows the performance of the four policies in the online simulation study. Here it is clear that TBL performs very well and evaluates the function close to its maximum relatively quickly leading to a small regret. Also LiF seems to converge, but, due to its continuous oscillations, the regret keeps increasing; a linear increase is expected for this policy. EF performs as expected; first it incurs high regret due to the exploration stage, after which regret is small in the exploitation stage. Given the relatively simple true reward function the exploitation stage often evaluates the reward function close to its maximum. The subsequent panels (left to right) show the performance of the distinct policies in terms of regret averaged over the 10^3 simulation runs for decreasing values of δ. Depicted are both the average regret over the 10^3 simulation runs, as well as their empirical standard errors (the confidence bands). Note that the standard errors for the offline evaluations increase heavily towards the higher values of the plot; this is caused by the fact that higher values of T become less and less likely in the offline evaluation.

As expected, for large values of δ a large number of observations is obtained (e.g., T is large), but the performance of the policies is severely affected by the extremely noisy evaluation of the true reward function. At $\delta = .5$ this results in an evaluation that provides hardly any information regarding the relative performance of the different policies. However, as δ decreases (and subsequently T decreases), we find a more and more clear ordering of the policies. To illustrate,

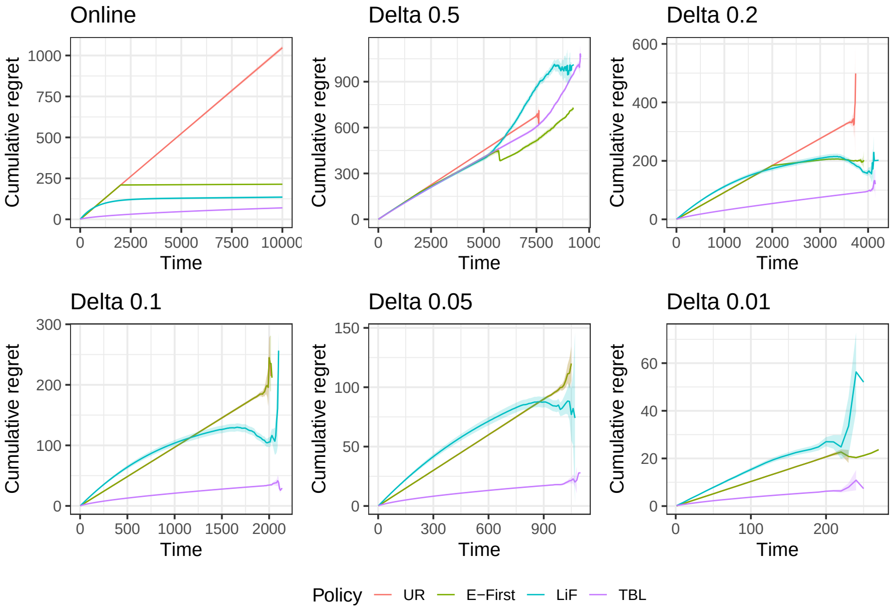


Figure 4.1: The results for the offline policy ranking simulation using a unimodal data generating model in terms of cumulative regret averaged over the simulation runs. The upper left panel shows the performance of the four policies for the online simulation. The other panels show the performance for the four policies for the offline evaluation with varying δ's. The lines are plotted together with their 95% confidence bounds. Duly note that the confidence bounds can be misleading, since for low δ's we have a low amount of T and also only a few repetitions left to average over (as compared to higher δ's). At the lower δ's, UR and EF have the same performance (because EF is not yet in the exploitation phase) and only EF is visible in the plot.

Table 4.1 presents the relative rank order of the policies in terms of lowest regret evaluated at $T = 1750$. The table makes clear that our proposed offline CAB policy evaluation method consistently ranks the policies that are being compared. Note that for smaller δ's, $T = 1750$ is not observed. One conclusion that can be drawn from this is that we need a relatively large offline data set for the method to work well.

The bimodal model Figure 4.2 shows the results for the bimodal, model. The first panel again shows the performance of the different policies in an online simulation. The figure displays a similar pattern for the UR and EF policies as before: steep linear regret for the UR policy, and the same regret in the exploration stage for EF, after which in the exploitation stage the regret is even higher; note that in this more complex case the optimum is not clearly found in the exploitation stages. Our implementation of TBL and LiF seem to have a comparable perfor-

Table 4.1: Table displaying the rank order of the four policies under scrutiny at $T = 1750$ for the online and offline evaluations for the unimodal model. Note that at this point by design UF and EF are in a tie. Further note that for small δ, $T = 1750$ is not observed.

Rank	Online	$\delta = .5$	$\delta = .2$	$\delta = .1$	$\delta = .05$	$\delta = .01$
1	TBL	TBL?	TBL	TBL	n/a	n/a
2	LiF	LiF?	LiF	LiF	n/a	n/a
3	EF/UR	EF/UR	EF/UR	EF/UR	n/a	n/a
4	EF/UR	EF/UR	EF/UR	EF/UR	n/a	n/a

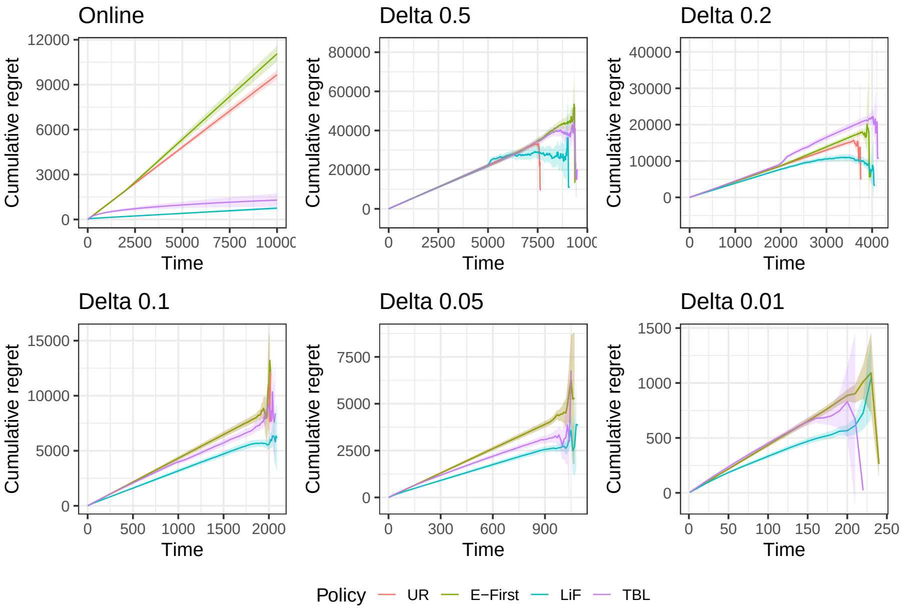


Figure 4.2: The results for the offline policy ranking simulation using a bimodal data generating model in terms of cumulative regret. The upper left panel shows the performance of the four policies for the online simulation. The other panels show the performance for the four policies for the offline evaluation with varying δ's. The lines are plotted together with their 95% confidence bounds. Duly note that the confidence bounds can be misleading, since for low δ's we have a low amount of T and also only a few repetitions left to average over (as compared to higher δ's).

mance; note that given our current specification of the model used for both TBL (Equation 4.2) as well as the implementation of LiF both policies are likely to get "stuck" in a local maximum thus incurring returning (linear) regret.

Table 4.2 again displays the relative rank ordering of the policies. Note that in this case again a clear – and correct – separation is visible between the EF and UR policies and the TBL and LiF policy: TBL and LiF are clearly preferred. This also

Table 4.2: Table displaying the rank order of the four policies under scrutiny at $T = 1750$ for the online and offline evaluations for the bimodal model.

Rank	Online	$\delta = .5$	$\delta = .2$	$\delta = .1$	$\delta = .05$	$\delta = .01$
1	LiF	TBL?	LiF	LiF	n/a	n/a
2	TBL	LiF?	TBL	TBL	n/a	n/a
3	EF/UR	EF/UR	EF/UR	EF/UR	n/a	n/a
4	EF/UR	EF/UR	EF/UR	EF/UR	n/a	n/a

carries through the offline evaluation. In any case, the offline evaluation would lead one to select a policy that performs relatively well on the current problem.

4.4.2 Offline parameter tuning

In this simulation, we will look at finding the learning parameters that will result in the highest cumulative reward for a given horizon. We start by retrieving benchmark values via an online simulation, which we then use to compare the results from the offline evaluation methods. In this study we will use both the delta and the kernel method. We will use the kernel method in this scenario to find out if the kernel method is able to provide us information about the relative difference between the performance of the different learning parameters, despite the fact that will we not be able to properly weigh the observations for the policies. Next to that we are interested in a performance estimate, we are interested in which setting of learning parameters will give us the best performance – there the (improper) use of the kernel method might be able to provide information. We simulate with $T = 10^4$ interactions and repeat it 10^4 times. For LiF, we are interested in finding an optimal amplitude A, so we run the simulation for a number of amplitudes between 0.002 and 0.2. Furthermore, we limit the actions of the LiF policy to be $[0, 1]$. The rest of the parameters for LiF are set as follows: $i = 50$, $\gamma = 0.1$, $\omega = \tau/i^3$ and $a_0 \sim \text{Uniform}(0, 1)$ (i.e. each repetition of simulation we set a different starting point). For TBL we are interested in finding an optimal prior precision P. We set $J = [5, 4, -4]$ and we run the simulations for number of different precisions between $P = \text{diag}(0.01, 0.01, 0.02)$ (i.e. low prior precision) and $P = \text{diag}(10, 10, 20)$ (i.e. high prior precision).

4.4.2.1 The desired true values of the learning parameters.

Figure 4.3 shows the results for the online parameter tuning scenario. We see different amplitudes for LiF and a low to high prior precision for TBL, expressed

[3] The constant $\tau = 2\pi \approx 6.2831$.

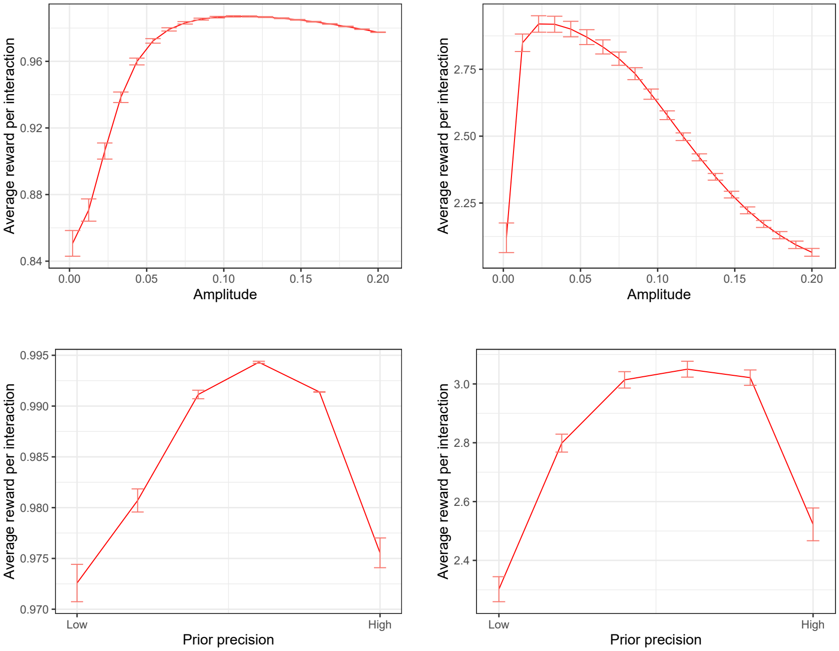


Figure 4.3: The results of testing LiF and TBL on online data on both data generating models. The top left panel shows the results for online simulation with LiF on Equation 4.5 and the top right panel shows the results with LiF for Equation 4.6. We find that an amplitude around 0.115 has the highest expected average reward per interaction for Equation 4.5 and an amplitude of 0.035 for Equation 4.6. The bottom left panel shows the results for online simulation with TBL on Equation 4.5. The bottom right panel shows the results with TBL for Equation 4.6. We see in both cases that a medium precision achieves the highest average reward per interaction.

in average reward per interaction for the two different data generating models. Here, we find that an amplitude around 0.115 has the highest expected average reward per interaction for the evaluation on Equation 4.5 and around 0.035 for the evaluation on Equation 4.6. Further note that amplitudes going towards 1 and higher will result in low average reward – intuitively this makes sense as the actions are in $[0, 1]$ and a higher amplitude will make the policy bounce around the action space too much. For TBL we see that we need to look at a prior precision that is neither high nor low. A too high precision will leave us exploring too little, and a too low precision will leads us to exploiting too late. These results will be used to validate the results of parameter tuning in the offline case later on.

4.4.2.2 Offline parameter tuning

We have implemented both Algorithm 3 and 4 in an offline evaluation setting for parameter tuning (for Algorithm 4 we return the rewards as generated by the kernel method and thereby not weighing the observation itself). With the online simulation we would generate a reward each time LiF and TBL suggested an action, but now we generate a dataset of random actions and rewards beforehand. The actions are uniform randomly sampled between 0 and 1 and the rewards are generated using the data generating models Equation 4.5 and 4.6. The dataset exists of 10^4 interactions and is repeated 10^4 times (each time having a different dataset). Then we run through the dataset row-by-row and evaluate the actions based on the evaluation algorithms. Again we use LiF with the following parameters: $i = 50$, $\gamma = 0.1$, $\omega = \tau/i$ and $a_0 \sim \mathrm{Uniform}(0, 1)$ and we are looking to optimise the amplitude A under these conditions. Once more, we explore the range of amplitudes between 0.002 and 0.2. For TBL we also matched the same setting as in the online case: $J = [5, 4, -4]$ and we explore different prior precisions between $P = \text{diag}(0.01, 0.01, 0.02)$ and $P = \text{diag}(10, 10, 20)$. Furthermore, for the delta method we look at $\delta = \{0.01, 0.1, 0.5\}$ and for the kernel method we set $h = 10^{(-4/5)}$ (suggested by Kallus and Zhou, 2018) and $Q = 1$ as the density function for the uniform distribution in our range is a constant.

4.4.2.3 Results

Figure 4.4 shows the results for the unimodal reward data generating model. We have also plotted the results for the online evaluation as it allows is to usefully compare the results. This was not possible for the kernel method as this skewed the scale of the plot too much and in place we have plotted a dotted line to showcase the optimal parameters from the online evaluation. What we find is that the kernel method fairly captures the tendency of different learning parameters. However, it does underestimate the average reward per interaction, as it is a factor 10 lower than in the online scenario. For the delta method we see that it is able to describe the tendency of the different learning parameters, as is especially the case for $\delta = 0.1$, but gives results with higher variance than the kernel methods. This method is, however, able to provide a similar estimate of the average reward per interaction of the original online scenario. A higher delta ($\delta = 0.5$) does not capture the data well enough, while a lower delta ($\delta = 0.01$) has too little data to provide a consistent estimate ($\delta = 0.1$ accepts on average 10^3 interactions while $\delta = 0.01$ only accepts 10^2 on average).

Figure 4.5 shows the results for the bimodal data generating model. For the kernel method, we again see that the average reward per interaction is underesti-

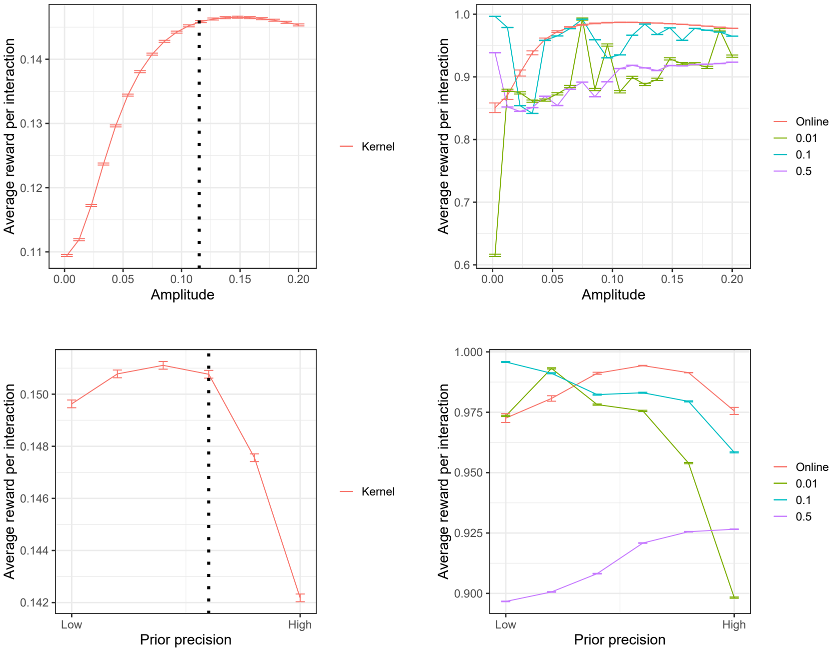


Figure 4.4: The results of testing LiF and TBL on offline data with the kernel (left) and delta (right) evaluation methods for the unimodal function. Top panels are LiF, bottom panels are TBL. For the delta method, we also plotted the parameters of the online evaluation. For the kernel method, plotting the online variant would skew the scale resulting in unclear view of its performance. To compare to the true parameters of the online evaluation, we highlighted the optimum with a dotted line.

mated. Here the evaluation of TBL with the kernel method has a higher variance, which can be due to the fact that TBL might be more sensitive to underestimating rewards than LiF. For the delta method, the most stable results are shown once more by setting $\delta = 0.1$.

Looking at these results, we see that in an effort to find the most optimal learning parameters, we would combine the results of both methods and their strengths to a) provide a good estimate of the expected average reward and b) provide a reasonable estimate of which learning parameters work best. Using these results, we would end up with setting learning parameters that are close to the most optimal learning parameters.

4.5 Offline evaluation using field data

After evaluating the performance of our method, we demonstrate how it can be used in practice. In collaboration with a company that (re-)sells products online

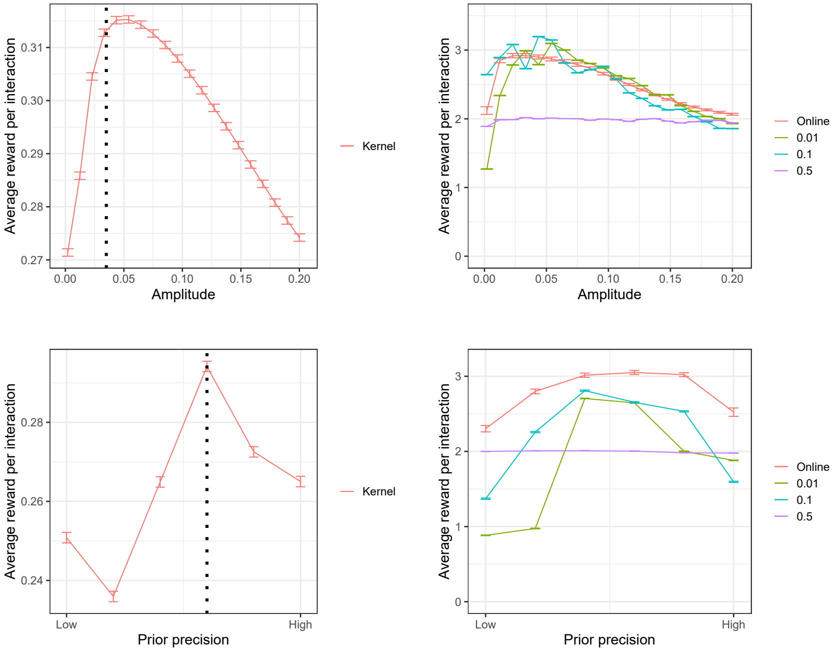


Figure 4.5: The results of testing LiF on offline data with the kernel (left) and delta (right) evaluation methods for the bimodal function. Top panels are LiF, bottom panels are TBL. For the delta method, we also plotted the parameters of the online evaluation. For the kernel method, plotting the online variant would skew the scale resulting in unclear view of its performance. To compare to the true parameters of the online evaluation, we highlighted the optimum with a dotted line.

by offering rebates for online stores we collected a dataset for offline CAB policy evaluation. The company negotiates deals with existing online stores, and offers a share of these deals to its customers. For example, an online store that sells sportswear can offer a 10% total discount to the rebate company. Next, the rebate company splits this discount between herself and the end-customer. The current practice is to split the discount 50-50: if a customer wants to buy a pair of shoes from a online store that cost $50\$$, the rebate company receives a total $5\$$ cashback from the online store and gives $2.5\$$ to the customer.

The company, however, does not know whether the 50-50 split actually maximizes their revenue. This gives rise to a CAB problem in which the action consists of the rebate percentage offered and the rewards consists of the revenues generated. Note that it is expected that a single maximum of this “split-revenue” function exists: passing a large part of the discount on to the customer likely leads to a large volume, but little revenue for the rebate company, while passing on a small discount likely leads to a smaller volume.

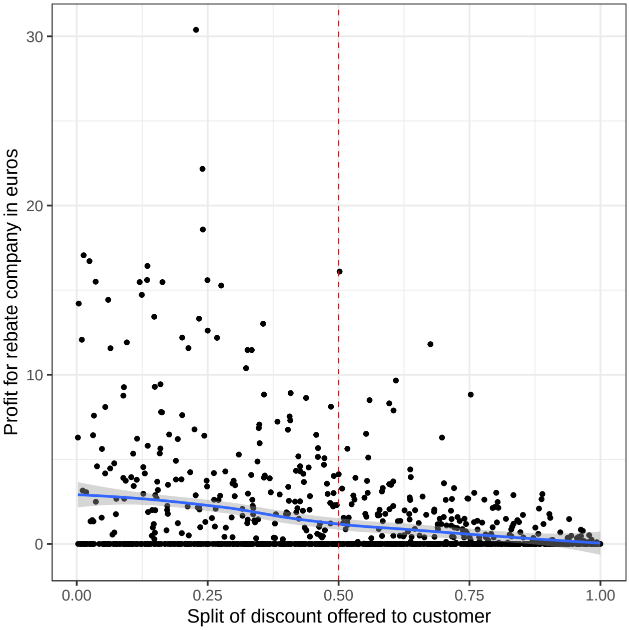


Figure 4.6: Revenue of the participating rebate company as a function of the proposed split.

We collected a data stream S in which the split proportion was selected randomly, $a_t \sim \text{Uniform}(0, 1)$, and used this data stream to evaluate different CAB policies. The offered discount to the customer was $y_t = 10a_t$ and the reward of the CAB policies are some function of the proposed discount, $r = f(y)$. Using **StreamingBandit** (Kruijswijk et al., 2020) we collected a field data set consisting of a total of 2448 data points (each consisting of a split a_t, and the actual revenue r_t). Figure 4.6 shows the revenue of the rebate company against these random splits.[4]

As in our simulation studies, we run an offline evaluation using the empirical data 10^3 times. We choose $\delta = 0.1$; this leads in expectation to around 500 valid observations which aligns roughly with the median number of visits the rebate company expects per newly introduced product (often the rebate offers are valid only for specific products and for a limited period). We have used the same starting values for the policies as in the simulation studies, except for EF, where we limit the exploratory phase to $N = 100$. Note that since we have no knowledge about the action with the highest expected reward, we can only compute the cumulative

[4] Note that the figure seems to favor very low splits; this might not be feasible in the long run. Our current analysis only considers a single shot purchase, and obviously a more reasonable and sustainable approach would include the customer lifetime value as a whole; most likely favoring higher customer discounts.

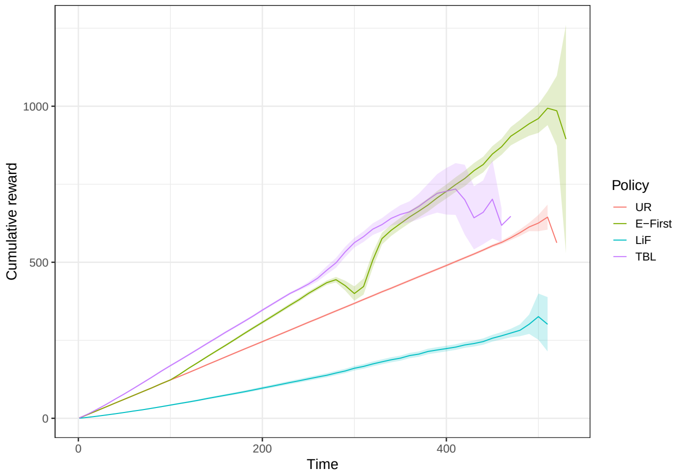


Figure 4.7: Average cumulative reward and confidence bounds of four different CAB policies using our offline evaluation method.

reward, as discussed before. Figure 4.7 shows the cumulative reward of the 4 policies also used in our simulation study. The evaluations show that EF (and in a sense TBL) obtain the highest cumulative reward. This analysis thus would encourage the company to use EF (or TBL) for the optimization of their revenue. This result is in-line with earlier studies that show that (for small-scale problems) simple heuristics often outperform asymptotically optimal policies (Kuleshov & Precup, 2014).

4.6 Conclusion

In this paper we proposed and empirically evaluated an offline policy evaluation method for the CAB problem that is inspired by the work by Li et al. (2011). The method works by sequentially running through logged events while comparing the logged action with the action suggested by a policy under evaluation. Next, the distance between the suggested action and logged action is compared and, if this distance is small enough, the data point gets accepted and evaluated. We showed that using this method the rank ordering of CAB policies stays relatively intact and hence that the method is potentially of use to select CAB policies for applied problems. In future studies, incorporating more reasonable competitors for the current policies would increase the strength of this method further. Our work, however, raises a number of questions.

Firstly, it seems that the ranking of policies is influenced by the complexity of the data generating model. Apparently, some policies are more robust to the noise introduced by our offline evaluation method than others and this interacts with our offline evaluation method; we will be looking for ways to quantify this difference and possibly correct for it.

Secondly, we would like to scrutinize further the effect of the tuning parameter δ, and perhaps quantify the behavior of distinct policies as δ decreases: as $\delta \to 0$ the actual (or online) behavior of the policy should surface and hence it is interesting to study the behavior of policies as a function of δ.

Furthermore, especially in relation to the introduced noise (first remark) and the scrutinization of δ (second remark), we did not consider a multi-dimensional action space, in which case the quality of the estimation of $\mathbb{E}(R_c)$ might be even more dependent on the chosen delta and the smoothness of the expected reward. Future research should ideally also take this into account.

Thirdly, there is currently no clear guidance on how to set δ. We must stress, however, that with our method, one can explore multiple values of δ for the offline evaluation and re-run it, since the choice of δ does not impact the collection of the training data. We have described the trade-off between a large and small δ before and we think it is useful to explore the robustness of the policies using multiple choices for δ. Nonetheless, a clear cut way of choosing δ would make this method more user friendly.

Finally, we would like to further study ways in which the noise introduced by the approximate evaluation of the true data generating model can be corrected. While currently the distance between the suggested and the logged action is not taken into account when updating the policy, we are experimenting with methods that update the parameters of a policy using a weight that is proportional to this distance (e.g., update the parameters of the policy using a discount that is dependent on $|a - \Pi(h_{t-1})|$ (as in Algorithm 3)). The weight could for example be calculated using the kernel from the kernel method. This would effectively result in a combination of our delta method with the kernel method. However, as identified in the first simulation study in Section 4.4.1, how to update policies using this weight is not trivial and dependent on the policy.

Additionally to the remarks of our current research, we have looked at another possibility for extending the method by Li et al. (2011) to use for the CAB problem. This would be to bin the continuous action space, such that we create an offline policy evaluator that is comparable to the method by Li et al. (2011). Firstly, our method can be considered a quite dynamic – and hence more continuous – form of binning (i.e., a bin centered around the evaluated action). Secondly, this also introduces a hyperparameter (similar to δ), determining the amount of bins, which would be (more) highly application dependent. Hence, we think that the solution proposed in this paper is more appropriate.

Furthermore, in our second simulation study we have looked into how offline evaluation can help with parameter tuning for dynamic policies with continuous treatments. We have compared two methods against online evaluation, where the combination of both methods provides the experimenter potentially with useful insights. Offline policy evaluation for the dynamic policies is beneficial for the design of future experiments where continuous treatments are concerned. Firstly, because employing simulation models to compare different policies will not always enable the researchers to capture the mechanisms of the real world. This is especially significant in cases where doing multiple field trials is expensive. And secondly, we are often interested in dynamic processes, where parameters should be able to adapt. Researchers are not always in the fortunate position to have a fundament of earlier literature to explore as was the case in Kaptein, van Emden, and Iannuzzi, 2016b, which in turn can lead to arbitrarily choosing parameters. Offline evaluation of dynamic policies allows researchers to compare multiples policies, different tuning parameters and settings of a policy. Nevertheless both discussed methods have weaknesses and they already show in this relatively simple problem: we consider no context and one-dimensional data. Two interesting research directions that can potentially be useful for future research are the use of the Direct Method (DM) (Dudík et al., 2014) and General Adversarial Networks (GAN) (Bai, Guan, & Wang, 2019). Both the DM and the GAN solutions fit models to the existing data to generate “similar” data – again using assumptions. Using those models, one can potentially evaluate any policy.

Effectively, we fear that we currently do not yet have the most appropriate methods to properly evaluate CAB policies offline. We content that the method and its empirical analysis presented in this paper provide an initial step towards the development of valid and effective *offline* policy evaluation method for CAB policies.

4

Chapter 5

A Tutorial on Using Sequential Allocation Procedures in Web-based Research

Abstract

In recent years scientists from various disciplines are increasingly interested in applying sequential treatment allocation procedures in experiments. Sequential allocation procedures allow researchers to capitalize on information that is collected while running their experiments. This information can potentially be used to make more informed decisions for treatment allocation in the remainder of the experiment, or, conversely, stop the experiment when sufficient data is collected. Simultaneously, the internet has facilitated researchers to conduct experiments quicker, easier and cheaper. Online platforms that offer services to conduct such experiments, typically also allow researchers to do some form of treatment allocation. However, despite the technological advances and the interest in sequential allocation procedures, most studies and platforms still use very simple designs (e.g., simple randomized allocation). In this article, we show how to deploy sequential allocation procedures in common front-end survey and experimentation platforms using a back-end software application called **StreamingBandit**. We demonstrate this by integrating **StreamingBandit** in **Qualtrics** (an online survey platform). We show how, when **StreamingBandit** is integrated to the desired platform, it is easy to adapt experiments to use different types of allocation procedures. We close off by showing a web-based field experiment that applied **StreamingBandit** and sequential treatment allocation. With this work, we enable researchers to reap the fruits of sequential decision making in future (web-based) experiments using standard tools.

Keywords: sequential allocation procedures, sequential decision making, web-based research, **StreamingBandit**, multi-armed bandit, experimental design

Submitted as Kruijswijk, J.M.A., Kaptein, M.C. A Tutorial on Using Sequential Allocation Procedures in Web-based Research.

5.1 Introduction

A critical component of designing experiments is deciding which treatment to allocate to which subject, in other words, deciding on the treatment allocation rule. Typically, treatment allocation is determined a priori and often it is done uniformly at random. For instance, in a traditional randomized controlled trial (RCT) with a 2-by-2 between-subjects design we randomly allocate subjects to one of the four conditions with a probability of $1/4$ (if the randomization ratio is equal over all conditions). However, throughout science, the use of sequential allocation procedures in the design of experiments is becoming more widespread (for a survey and examples, see e.g., Bouneffouf & Rish, 2019; Clement et al., 2015; Kaibel & Biemann, 2019; Press, 2009). Sequential allocation procedures make treatment allocation decisions while running the trial, often in a one-by-one fashion. Using sequential allocation procedures allows researchers to capitalize on information gathered from earlier treatment allocations and those results to use in future treatment allocations. For example, a researcher may be interested in estimating the effect of two different conditions where possibly the variance between those conditions greatly differ (i.e. the observations in one condition have a higher variance than in the other), which can in turn result to unreliable estimates. Using a sequential treatment allocation procedure would allow to dynamically allocate conditions based on the observed variance, and allocate more treatments to the condition with the highest observed variance to increase estimation precision (Kaptein, 2014). Another example might be a researcher that is interested in applying early stopping rules when certain conditions are met or when constraints fail (see e.g., Berry, 2006).

The possibilities of using sequential allocation procedures are enormous. Research has already shown a hint of the potential, which include the design of adaptive clinical (medical) trials (Berry, 2012; Coffey & Kairalla, 2008; Durand et al., 2018), behavior modeling of human decision making in patients with mental disorders (Bouneffouf, Rish, & Cecchi, 2017), the study of social phenomena that suffer from noisy signals (e.g. treatment heterogeneity and confounders) (Kaptein et al., 2017), and influence maximization in social network analysis (Vaswani et al., 2017). These examples are a small portion of the studies that benefitted from using sequential allocation procedures.

Parallel to this development, the surge of the internet has enabled researchers to do experiments quicker, easier and cheaper. Websites like Amazon's Mechanical Turk (**MTurk**) facilitate the crowdsourcing of subjects on a large scale to use for conducting experiments (Amazon, 2012). Next to that, websites such as **SurveyMonkey** and **Qualtrics** ease the process of conducting a survey (Qualtrics, 2005; SurveyMonkey, 1999). They both offer extensive software to create surveys (amongst others) and to deploy them easily. Both **MTurk** and the likes of **SurveyMonkey** and **Qualtrics** allow current research to be more flexible and enable research to be deployed to a wider audience than possible in traditional research. One downside of these websites is that, although they facilitate conducting surveys and simple experiments in an easy way, they are in some regards limited when it comes to applying different types of (sequential) allocation procedures. For example, **Qualtrics** only allows simple (uniform) randomization within its survey flow. The possible benefits that sequential allocation procedures provide in the design of experiments might thus not always be trivial to implement in these scenarios.

In this article, we want to show how to incorporate sequential allocation procedures for experiments in web-based studies where such features do not yet readily exist. We do this using our own developed software called **StreamingBandit** (see our full paper for details on the software (Kruijswijk et al., 2020)). **StreamingBandit** was designed to create sequential allocation procedures and to easily integrate with existing software and websites. As a running example, we will demonstrate its use with **Qualtrics**, but note that **StreamingBandit** can be integrated into any platform that allows for customized web service integration (explained in more detail in Section 5.3).

To the best of our knowledge, our approach is novel in that it provides researchers a flexible tool to apply any type of (sequential) allocation procedure. Other approaches have tried to create systems or frameworks that allow easy deployment of experiments in web-based experiments. These approaches are limited when compared to **StreamingBandit** in the sense that they only 1) allow for a limited set of allocation procedures, 2) are (black-box) optimization software tuned towards optimizing outcomes of experiments (e.g., optimizing click-through rates for advertisements on the web), or 3) require more programming knowledge. For example, **PlanOut** (Bakshy, Eckles, & Bernstein, 2014), **AE** (or **AX**) by Facebook (Bakshy et al., 2018), **APONE** (Marrero & Hauff, 2018) and **DEXPER** (Williams et al., 2017) are frameworks and software to deploy experiments on the web. **DEXPER** is a proof-of-concept web service for dynamic experimentation for (educational) platforms, but it is mostly geared towards optimization of outcomes and has only implemented a limited set of allocation procedures. **APONE** is a web service to simplify running A/B tests, but it therefore is limited to only allocating treatments uniformly at random. **AX** is an adaptive experimentation platform designed in Python, but it is also geared towards optimization of sequential applications and is not built with a framework to translate allocation procedures such as **StreamingBandit**. **PlanOut**, a framework and package for developing adaptive experiments, is the most flexible of these examples, as it allows its users to implement any type of allocation procedure, but it is a Python package rather than a complete application that uses a REST architecture design and thus requires more extensive knowledge of programming than **StreamingBandit** does – i.e. it can be used when programming whole applications but it is less easy to use in combination with existing (survey) tools. Next to these general frameworks and software, some research has been done on implementing allocation procedures in **Qualtrics** specifically. For example, Weber (2019) shows how to implement a discrete choice experiment (DCE) and Dropp (2014) shows how to implement a conjoint analysis design. However, none of the works focus on supplying a generic toolkit to implement any type of allocation procedure. This is exactly what we provide in this paper, a flexible framework and toolkit for implementing any type of sequential allocation scheme in a web-based format.

The article is structured as follows. In Section 5.2, we will introduce a formalization of sequential allocation procedures and show a few examples of existing and practically applicable sequential allocation procedures to illustrate our formalization. Here we also make clear that the traditional, uniform random allocation is just a special case of a sequential allocation procedure in which the allocation itself is independent of the intermediary results. Then in Section 5.3 we will introduce and illustrate the basic usage principles of **StreamingBandit**, which is the back-end software that we use for the actual treatment allocation logic. In Section 5.4, we will show how to apply sequential allocation procedures by using **Qualtrics** as a front-end to **StreamingBandit**. Again, this choice of front-end is arbitrary. And concluding, in Section 5.5, we show an example of a recently conducted experiment that used a sequential allocation procedure – this example shows how some experimental designs benefit from using a sequential allocation procedure, as this experiment would have been virtually impossible without using one. Note that this article is intended for researchers that are interested in benefitting from sequential allocation procedures. The applications are not limited to the examples shown in this article – we are merely showing a hint of all the possibilities. Also note that to be able to follow the article, we assume basic familiarity with the Python programming language, some statistics knowledge and working knowledge of **Qualtrics**. Despite this, the article might still provide useful information on a) sequential allocation procedures and b) deploying them in web-based research for those who are less familiar with the prerequisites. All the code and exported **Qualtrics** experiments (QSF files) are available on GitHub at https://github.com/Nth-iteration-labs/sequential-experimentation-on-the-web.

5.2 Formalization of sequential allocation procedures

Sequential allocation procedures are commonly researched under the heading of the multi-armed bandit (MAB) problem. The classical MAB problem can be described as follows: subjects are assigned to one of a multitude of treatments (also called arms or actions) in a one-by-one fashion with the intent to, during the experiment, maximize an outcome that is possibly affected by the treatments (also called reward) (Berry & Fristedt, 1985; Whittle, 1980). Note here that the prototypical objective of the MAB problem contrasts that of the RCT. The MAB problem originally focusses on maximizing the outcome over all allocations, while in the RCT framework we are interested in finding the treatment or arm with the highest effect. Additionally, while in the traditional RCT framework we are used to thinking about assigning subjects to one of a number treatment arms, in the MAB problem we think of assigning one of the arms to the (sequentially emerging) subjects. Despite this difference, the MAB formalization can help us to frame experiments as a sequential decision problem. The vast MAB literature offers us strategies to address these problems. These strategies typically try to balance the learning of the outcomes (called exploration) and selecting the treatment that has the highest expected outcome (called exploitation). This is also known as the exploration-exploitation trade-off. As said before, traditional experiments are used to estimate the effect of an arm and not necessarily to maximize the overall outcome within the experiment. Incorporating MAB allocation procedures in clinical trials can therefore be challenging when confidence about the estimation of a statistic is a high priority – for example, traditionally MAB allocation procedures have severe limitations in terms of statistical power. However, recent research has been trying to overcome this (for further discussion see e.g., Villar et al., 2015; Williamson et al., 2017).

For the remainder of the paper, we use the formal definition of the MAB problem to construct the allocation procedures for the experiments. To give a formal definition, the MAB problem can be described as follows: at each time $t = 1, \ldots, T$ (also called interactions) we choose an action a (i.e., treatment) from a set of possible actions $\mathcal{A}$. After choosing action $a \in \mathcal{A}$ we observe a reward r_t. The objective in the canonical MAB problem is to find a policy $\Pi : \mathcal{D}_{t-1} \to a_t$ that maps all historical data $\mathcal{D}$ (consisting of tuples of $\{(a_1, r_1), \ldots, (a_{t-1}, r_{t-1})\}$) to the next action a_t and tries to maximize the cumulative reward $R_C = \sum_{t=1}^{T} r_t$. Here we are less interested in policies that perform well, but rather we treat a policy as a formal sequential treatment allocation procedure and demonstrate how to implement flexible policies. A generalization of the MAB problem is the contextual

MAB (CMAB) problem, where covariates or features of the subjects can also be taken into account (in the so-called context), which possibly further influence the reward outcome. When we consider the CMAB problem, we observe context x_t before choosing an action a_t. A policy is then defined as $\Pi : (x_t, \mathcal{D}_{t-1}) \rightarrow a_t$ ($\mathcal{D}_{t-1}$ is now composed of $\{(x_1, a_1, r_1), \ldots, (x_{t-1}, a_{t-1}, r_{t-1})\}$). The MAB literature explores many different policies, often with the goal of maximizing reward. A few popular examples include Upper Confidence Bound (UCB) methods (Auer et al., 2002; Lai & Robbins, 1985; Li et al., 2010) and Thompson sampling (Chapelle & Li, 2011; Thompson, 1933).

Using the formalization, the RCT can be interpreted as a potential solution to a MAB problem: patients typically arrive in a one-by-one fashion and are randomly assigned (possibly based on certain conditions) to a treatment condition. The treatment allocation in the RCT framework can be formulated as follows (and is just one of many possible policies):

$$\Pi(\mathcal{D}) \coloneqq a_t = \begin{cases} \text{Rand}(a \in \mathcal{A}) & \text{if } t < \epsilon \\ \arg\max_a(\hat{\theta}^a) & \text{otherwise} \end{cases} \tag{5.1}$$

5

where $\hat{\theta}^a$ is the estimate of the effect of an action. In the first interactions we randomly select an action with $\mathcal{A} = \{\text{treatment}, \text{control}\}$ (i.e., the exploration phase). We do this until the number of interactions reaches a threshold ϵ (with $0 < \epsilon < T$). Then after exploring for ϵ interactions, we always choose the treatment with the highest expected outcome (i.e., the exploitation phase). In the MAB literature, this policy is known as ϵ-first (Sutton & Barto, 2011).

Another example of a MAB policy is Thompson sampling, which works as follows: based on prior information and data from the current experiment, select an action a_t proportionally to probability of being optimal. Typically, using a posterior distribution, it suffices to sample a θ'^a from $P(\theta|\mathcal{D})$ for each action and select the action with the highest θ'^a. Thompson sampling performs asymptotically optimal (for more details, see e.g. Agrawal and Goyal (2012)) and is easy to implement if it is easy and computationally efficient to sample from $P(\theta|\mathcal{D})$ (e.g. we have a closed-form expression for the posterior via a conjugate prior). If it is not easy to sample from the posterior, it can for example be replaced with a bootstrap distribution (Eckles & Kaptein, 2019). Formally, this policy can be written as:

$$\Pi(\mathcal{D}) \coloneqq a_t = \arg\max_a(\theta'^a) \tag{5.2}$$

where θ'^a is a single draw from the posterior $P(\theta|\mathcal{D}_{t-1})$ for arm a. To be able to easily conduct experiments that utilize sequential allocation procedures on the web, we built software that uses the framework of the MAB formalization, as it gives us a rigid structure and a vast literature to establish these procedures. In the next section, we introduce this software.

5.3 StreamingBandit

StreamingBandit is a Python application that is designed with the purpose to easily allow researchers to experiment with (sequential) treatment allocation procedures in a web environment (Kruijswijk et al., 2020). Figure 5.1 shows a visual overview of how **StreamingBandit** could be applied in a web-based experiment. The chosen experimentation front-end communicates with **StreamingBandit** to implement the allocation procedure in the experiment. In turn the experimentation front-end serves the experiment to the subjects behind their computer and returns the outcome of the experiment to **StreamingBandit** to update the sequential allocation procedure.

Before we dive into the details of how the **StreamingBandit** back-end can be implemented in a front-end service such as **Qualtrics** (see next Section 5.4) we will first give a small demonstration of how **StreamingBandit** works internally. For detailed installation instructions for **StreamingBandit**, please refer to the paper (Kruijswijk et al., 2020) or the GitHub repository https://github.com/nth-iteration-labs/streamingbandit. For the remainder of the paper we assume that **StreamingBandit** is installed.

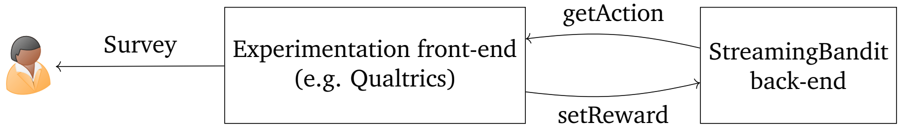


Figure 5.1: Visual overview of how the **StreamingBandit** back-end is integrated with an experimentation front-end.

5.3.1 Implementation details

StreamingBandit is built based on the observation that any bandit policy can be split into two distinct functional steps: 1) choosing the treatment and 2) updating the history (or parameters). These two steps are implemented into two so-called web API endpoints. Web API endpoints allow a user to communicate with software over the internet using the HTTP protocol (the same you use when entering websites in your browser, clarifying examples will be shown later). These two core API endpoints in **StreamingBandit** allow the user to select an action (the first API endpoint is called getAction) and return a reward (the second API endpoint called setReward). The advantage of **StreamingBandit** is that once it is integrated as a back-end to communicate with whatever front-end (e.g. **Qualtrics**), it is straightforward to change the sequential allocation procedure from within **StreamingBandit** – you are not tied to only one type of experimental design. The provided codebase enables switching from a traditional RCT sampling scheme to Thompson sampling, amongst others, with just a few lines of Python code.

We can more formally define the core functionality of **StreamingBandit**; the decision step, which results in the getAction API endpoint and the summary step, which results in the setReward API endpoint. The two steps formally look as follows:

1. The *decision* step: using x_t and θ_{t-1}, and often using some (statistical) model relating the actions, the context, and the reward which is parametrized by θ_{t-1}, a next action a_t is selected. Making a request to **StreamingBandit**'s *getAction* API endpoint returns an object containing the selected action.

2. The *summary* step: θ_{t-1} is updated using the new information $\{x_t, a_t, r_t\}$. Thus, in this step, $\theta_t = g(\theta_{t-1}, x_t, a_t, r_t)$ where $g()$ is some update function. Making a request to **StreamingBandit**'s *setReward* endpoint containing an object including a complete description of $\{x_t, a_t\}$, and the reward r_t, allows one to update θ_t and subsequently influence the actions selected at $t + 1$.

These two steps are implemented in Python code that is automatically stored in a custom database. The stored code for each experiment will be loaded into an internal *Experiment* class. This class contains several functions that help to store and retrieve parameters from a database – which relieves some burden on the user. The most important of these functions will be discussed in the examples in Section 5.3.2 and Section 5.4.

Oftentimes the statistical models or parameters θ that are used in allocation procedures are (based on) common estimators such as the mean and proportion. For example in ϵ-first, we want to select an action with the highest expected reward in the exploitation phase. We look at the average of all historical rewards and select the action with the highest mean. The most widely used estimators – such as the mean, variance and others – are already implemented in **StreamingBandit** in separate classes (in the `base` module, examples will be shown later) so most users will not have to implement estimators themselves.[1] The estimators share similar functionality (e.g. they all have an `update` function), which makes it easy to switch from one estimator to another. Additionally, the software allows to easily save multiple estimators, for instance when you need an estimator for each different condition, in a list (using the built-in `base.List` class). This list also contains some

[1] For complete information on the implemented estimators and models, please refer to the documentation at: https://nth-iteration-labs.github.io/streamingbandit/.

functionality to easily choose a random action, or the action with the maximum value of the estimator.

5.3.1.1 Ensuring computational scalability

To ensure that the implemented procedures in **StreamingBandit** do not suffer from a decline in performence in cases where experiments have a lot of data, we assume that at the latest interaction t all the information that is necessary to select a new treatment can be summarized using a limited set of parameters θ_t. Effectively, all the prior data, $\mathcal{D}_{t-1}$ are "summarized" into θ_t. This choice makes that the computations are bounded by the dimension of θ and the time required to update θ instead of growing as a function of t. Note that this effectively forces users to implement an online policy; the complete dataset $\mathcal{D}_{t-1}$ is not revisited at subsequent interactions (also called streaming (or online) updating, see e.g. Ippel, Kaptein, and Vermunt (2016a), Michalak et al. (2012)).

5.3.2 Example allocation procedure in StreamingBandit

To give a small demonstration of what an experiment looks like in **StreamingBandit**, let us take a look at code for a 2-by-2 between-subjects design with two different drugs A and B. Our four conditions would be as follows: A, B, A&B and None. For $N = 100$ subjects, we want to randomly allocate each of these four treatment conditions. In traditional experiments, after $N = 100$ we typically select the action that has the highest expected reward (i.e. if we were to select the best treatment condition after the experiment). In **StreamingBandit**, our getAction code would look as follows:[2]

```
# Retrieve parameters from database
propl = base.List(self.get_theta(key="treatment"), base.Proportion,
↪  ["A", "B", "AB", "None"])
# Check if exploration phase has ended
if propl.count() > 100:
    # If so, select the action with highest expected reward
    self.action["treatment"] = propl.max()
else:
    # If not, select a random action
    self.action["treatment"] = propl.random()
```

[2] Note that we do not need to import any packages. **StreamingBandit** imports a standard set of packages automatically when executing the code.

In the first line, we get our current parameters from the database using the `self.get_theta` function from the *Experiment* class. The parameters are put in a `base.List` class, to enable some functionality that we would like to have. We tell this list that the parameters are proportions (of successes) using `base.Proportion` and that we have four different options (our conditions). Then we count how many times we have already given out an action using the `propl.count()` function. If this count is higher than 100, we pick the treatment with the highest proportion of successes. If this count is lower than or equal to 100, we randomly pick a treatment. Now effectively any front-end service can use the getAction API endpoint as follows:

```
http://HOST:8080/getaction/<exp_id>?key=<key>
```

where `<exp_id>` and `<key>` are unique strings provided by **StreamingBandit** and `HOST` is the location of the installed **StreamingBandit** instance (e.g. the IP address of your server). Putting this HTTP call into our browser, we receive the following the response in the format of a JavaScript Object Notation (JSON): `{"action": {"treatment": "AB"}, "context": {}}`. **StreamingBandit** communicates using JSON as it is a widely accepted standard on the web and is also human-readable and thus readily applicable in many situations. In this example response, **StreamingBandit** has now randomly selected action AB.[3]

Then, when data has been collected for this subject, we want to update the parameters (i.e. update if the experiment was a success or not) inside **StreamingBandit** and we do this with the following code for the setReward endpoint:

```
# Retrieve parameters for the played action from database
prop = base.Proportion(self.get_theta(key="treatment",
↪   value=self.action["treatment"]))
# Update the proportion with the built-in update function and the
↪   reward
prop.update(self.reward["value"])
# Update the parameters in the database
self.set_theta(prop, key="treatment",
↪   value=self.action["treatment"])
```

In the first line, we select only the parameters from the database from the treatment that has been chosen. In our case, this will be AB for the first time and this is done using the `value` argument of the `self.get_theta` function. Then we put this in a `base.Proportion` class. We use the built-in `update()` function to incorporate

[3] Due to randomness it is possible that users receive a different treatment.

Table 5.1: An example of how the treatment allocation with ϵ-first and Thompson sampling can play out. For ϵ-first, after 100 interactions, the action with the highest observed reward is selected. For Thompson sampling, after 100 interactions, the policy keeps selecting actions proportionally to the probability of begin optimal.

	ϵ-first		Thompson sampling	
Interaction	a_t	$P(a_t)$	a_t	$P(a_t)$
1	A&B	0.25	A&B	0.25
2	A	0.25	A&B	0.3
3	None	0.25	B	0.25
⋮	⋮	⋮	⋮	⋮
101	B	1	B	0.9
102	B	1	B	0.9

the returned outcome (i.e. reward) and in the final line, we put our updated parameters back in the database. We can use the setReward API endpoint using the following HTTP call:

```
http://HOST:8080/setreward/<exp_id>?key=<key>&action={"treatment":"⌟
↪  AB"}&reward={"value":1}
```

Calling this HTTP call results in the following JSON output:

```
{"status": "success", "action": {"treatment": "AB"}, "context": {},
↪  "reward": {"value": 1}}}.
```

StreamingBandit returns a success status in the resulting output, we know that **StreamingBandit** has successfully ran the setReward code and updated our parameters. Table 5.1 illustrates how the allocated treatments can potentially play out (in this case assuming that treatment B will have the eventual highest expected reward). The probability of selecting an action is 0.25 in the first 100 interactions, after which one action is played forever (and thus the probability of selecting that action is 1).

5.3.3 From ϵ-first to Thompson sampling

Using the first implemented example, we can demonstrate one of the core strengths of **StreamingBandit**: changing the allocation procedure without even touching the front-end application. We could for example change from randomly uniform allocation to Thompson sampling by changing our getAction code to read the following:

```
# Retrieve parameters from the database and store them in a
↪  Thompson sampling class
propl = thmp.BBThompsonList(self.get_theta(key="treatment"),
↪  ["A","B","AB","None"])
# Select an action using the built-in Thompson sampling function
self.action["treatment"] = propl.thompson()
```

In the first line, we again get our parameters from the database and we insert it in our Thompson sampling class. Then we use the built-in `thompson()` function to sample one of the four treatments with a Thompson sampling scheme. Interestingly, our setReward code does not even need to be changed as we are updating the parameters in the same way. Thus by changing only a few lines of code, we have implemented a true sequential allocation procedure – the front-end experimentation software needs no adaptations. Table 5.1 illustrates, in the last two columns, how the allocated treatments can play out. Thompson sampling always selects actions proportionally to their probability of being optimal. As it might turn out that action B is optimal, it will select that action more often than others.

This section has shown a hint of the capabilities of **StreamingBandit**. The functionality of **StreamingBandit**, however, goes beyond implementing the getAction and setReward calls – such as testing and simulating the intended behavior of the allocation procedures. The complete range of capabilities are discussed in length in the original paper (Kruijswijk et al., 2020). For the remainder of the examples that will be shown in the next sections, we will cover extra details of the software where needed.

5.4 Implementation examples of allocation procedures

In this section we will describe how **StreamingBandit** makes it easy for researchers to implement complex treatment allocation procedures in web-based experiments. We focus on the use of **Qualtrics** as a front-end – although as said before, **Qualtrics** serves merely as an example and using **StreamingBandit** is not limited to only **Qualtrics**. **Qualtrics** is a well-known online tool that allows users to easily develop and deploy surveys (amongst others). Internally, **Qualtrics** offers different tools to handle the randomization of a survey. It is, however, limited to only doing a random coin flip and using conditional statements (i.e. "if coin flip is 0 go A, if it is 1 go B", or "if male go A, if female go B") and does not feature any extras. In the remainder of this section, we will show how to easily integrate

StreamingBandit in **Qualtrics** to enhance your experiments. We use **Qualtrics** as an example as it is very intuitive to use. To demonstrate a wide range of features, we implement the following procedures:

1. Between-subjects design
2. Between-subjects design with balancing
3. Within-subjects and mixed design
4. Thompson sampling for increasing estimation precision

To connect **StreamingBandit** with **Qualtrics** for all the examples, we use a few tricks inside **Qualtrics**. **Qualtrics** provides so-called *Web Service* blocks within their *Survey Flow*. These *Web Service* blocks allow a user to input any HTTP URL (such as the getAction and setReward calls) and consequently to catch the response and save it inside the survey – this is done via the so-called *Embedded Data*. Based on the value stored in the *Embedded Data*, which then represents the treatment condition, the survey can then be branched to different variants of the surveys. In the examples below we show this is done exactly.

5.4.1 Between-subjects design

Although a between-subjects design is feasible to do solely within **Qualtrics**, we show how it is done using **StreamingBandit**. In this case, we only need to use the getAction call, as we are not yet interested in integrating the results for the sake of the allocation of the conditions. Figure 5.2 shows the complete steps in pictures and code, which we explain step by step.

1. (see Figure 5.2a) The code for the getAction call is only one line of code in which we select a random action. Using the following HTTP call:

   ```
   http://HOST:8080/getaction/<exp_id>?key=<key>
   ```

   we will receive the following result:

   ```
   {"action": {"treatment": 2}, "context": {}}
   ```

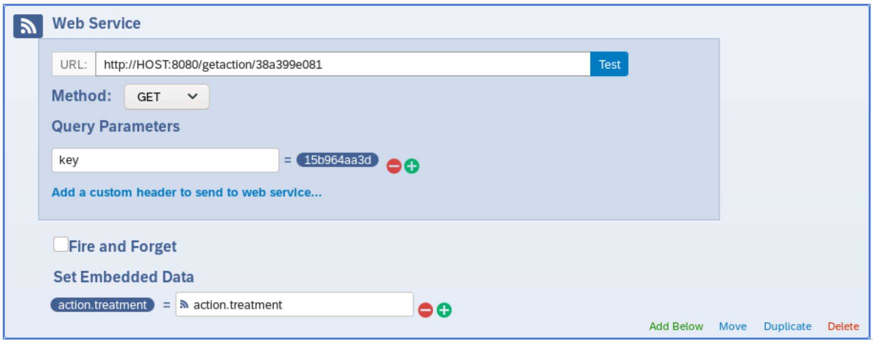
2. (see Figure 5.2b) Then we use the *Web Service* block within the *Survey Flow* of **Qualtrics** and fill in the details as shown in Figure 5.2b. We fill in the right information for the HTTP call (where `HOST` is the IP address of the **StreamingBandit** instance) using the supplied `key` and `exp_id` from **StreamingBandit**. We save the returning `action.treatment` as *Embedded Data* (which captures the `"action": {"treatment": 2}` part of the result of the HTTP call).

3. (see Figure 5.2c) We can then use the saved `action.treatment` value to branch our survey. Inside the *Survey Flow* we can insert a *Branch* and branch to a certain set of questions based on the treatment that is stored in the *Embedded Data*. Figure 5.2c shows how this is done for condition 1.

4. (see Figure 5.2d) And of course this must also be done for condition 2, which is shown in Figure 5.2d.

```
# Select a random action
self.action["treatment"] = random.randint(1,2)
```

(a) Code for selecting just a random action for a between-subjects design with no stopping rules.



Web Service

URL: http://HOST:8080/getaction/38a399e081

Method: GET

Query Parameters

key = 15b964aa3d

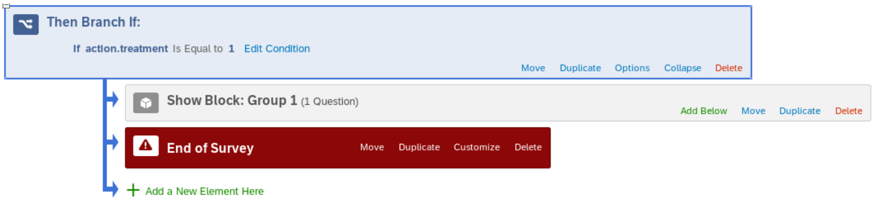
Add a custom header to send to web service...

□ Fire and Forget

Set Embedded Data

action.treatment = action.treatment

(b) Shown here is a *Web Service* block that calls the getAction API call of our **StreamingBandit** instance. We insert the right `key` and `exp_id` and save the returning `action.treatment` as *Embedded Data*.



Then Branch If:

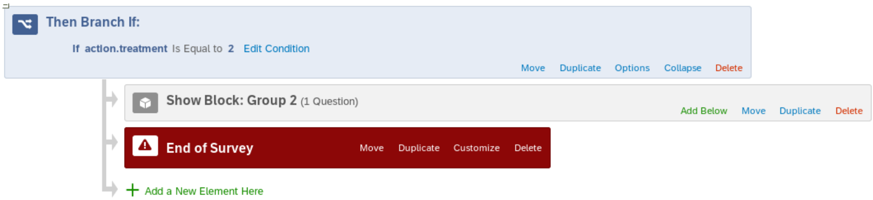
If action.treatment Is Equal to 1 Edit Condition

Show Block: Group 1 (1 Question)

End of Survey

Add a New Element Here

(c) An example of how to branch your survey using the *Survey Flow* of **Qualtrics**. We use our saved *Embedded Data* variable `action.treatment` as a conditon. If it returned 1, we go to the survey block of condition 1.



Then Branch If:

If action.treatment Is Equal to 2

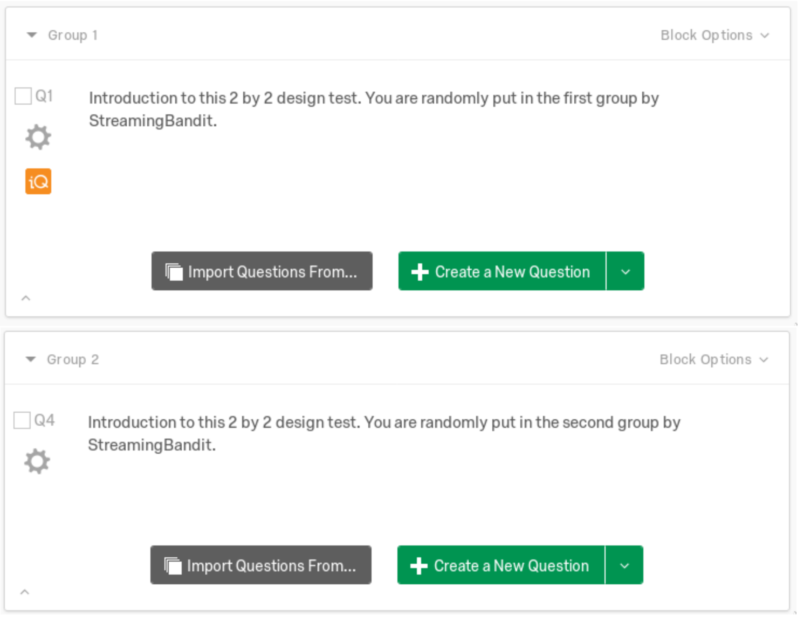
Show Block: Group 2 (1 Question)

End of Survey

(d) An example of how to branch your survey using the *Survey Flow* of **Qualtrics**. We use our saved *Embedded Data* variable `action.treatment` as a conditon. If it returned 2, we go to the survey block of condition 2.

Figure 5.2: The flow for a between-subjects design implementation in **Qualtrics** using **StreamingBandit**.

In the survey itself we can now add two blocks of questions, each for the separate conditions. In Figure 5.3 an example introduction text is shown and Figure 5.4 shows how the subject perceives the introduction text for this survey.



Group 1

Q1 Introduction to this 2 by 2 design test. You are randomly put in the first group by StreamingBandit.

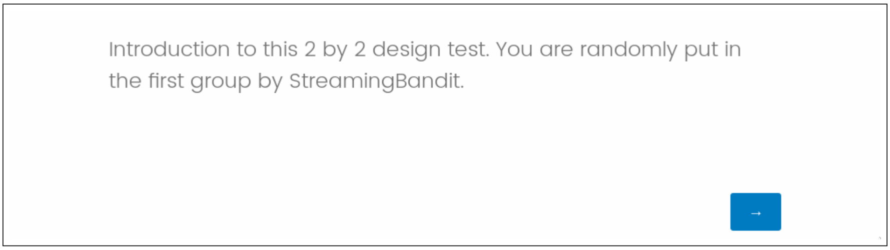
Group 2

Q4 Introduction to this 2 by 2 design test. You are randomly put in the second group by StreamingBandit.

Figure 5.3: Examples of two blocks of questions for different conditions in the survey, which can be linked in the *Survey Flow* to the different branches.

Table 5.2: An example of how the treatment allocation can occur for the between-subjects design for the subjects that are sequentially entering the survey.

Subject	Selected treatment
1	1
2	1
3	2
4	1
⋮	⋮



Introduction to this 2 by 2 design test. You are randomly put in the first group by StreamingBandit.

Figure 5.4: The eventual introduction text shown to a subject doing the survey.

Table 5.2 shows an example of how the conditions are allocated to the subjects that are taking the survey. Subjects are randomly put into either condition 1 or condition 2 for as long as the survey is conducted. Using these steps we have now implemented a between-subjects treatment allocation procedure. If need be, the conditions can be changed and extended – for example, the random.choice function can be used to choose between two string conditions.

5.4.2 Between-subjects design with balancing

When sending out a large batch of surveys with multiple treatment conditions it can happen that for one (or multiple) conditions there are proportionally far fewer responses. One way to combat this is to send out another batch of surveys, but this takes time and does not ensure that the responses will be balanced in the end. Effectively, if we feel that the order in which the subjects arrive is already random, we can allocate the conditions in a less or non-random order to balance out the number of fulfilled surveys. To then make sure that we have a balanced number of fulfilled surveys, we have to keep a count of the number of finished surveys for each condition and then allocate conditions based on the number of finished surveys per condition (i.e. the surveys with less finished surveys will get more subjects allocated). This is already a sequential design in the sense that the future allocations depend on the previous ones. Figure 5.5 shows the complete steps, which we explain step by step.

1. (see Figure 5.5a) The getAction code will involve a little bit more code, as we need to keep count of the number of survey requests we have had and the number of finished surveys we have had. In this example we will continue with two conditions (i.e. 1 and 2). In the first two lines, we get the counts for the number of requests and number of finished surveys from the database. Then we first check if the number of total requests (so the count for 1 and 2 combined) is smaller than our preset $n = 1000$. If this is the case, we randomly select a condition. If this is not the case, we select the condition with the lowest count (after the `else`-statement) using the built-in `min`-function. After we have selected an action, we make sure to increment the request count for the selected condition as shown in the last three lines.

2. (see Figure 5.5b) As we want to keep track of the number of finished surveys as well, we will call the setReward API call at the end of the survey. In the code as shown in the block in Figure 5.5b we get the count data for the specific condition from the database, update it and store it again in the database. Then we can use the following HTTP call to set the reward:

   ```
   http://HOST:8080/setreward/<exp_id>?key=<key>&action={"treatmen⌟
   ↪  t":1}&reward={"finished":1}
   ```

   The resulting response shows a success:

   ```
   {"status": "success", "action": {"treatment": 1}, "context":
   ↪  {}, "reward": {"finished": 1}}
   ```

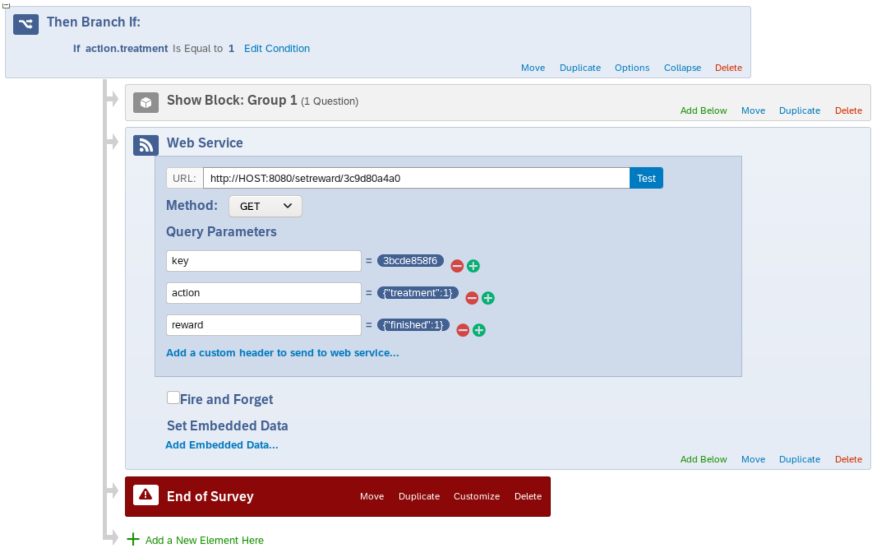
3. (see Figure 5.5c) The only thing that is left to do is to adapt the *Survey Flow* to send a call to the setReward call when a survey is finished. Figure 5.5c shows how this is done for the first condition. For the second condition this is equivalent but with the specific numbers changed.

```
# Set the number surveys before balancing
n = 1000
# Retrieve both the number of handed out surveys and the number
↪  of fulfilled surveys from the database
request_countl = base.List(self.get_theta(key="request_count"),
↪  base.Count, ["1", "2"])
fulfill_countl =
↪  base.List(self.get_theta(key="fulfilled_surveys"),
↪  base.Count, ["1", "2"])
# For the first n surveys, give a random condition
if request_countl.count() < n:
    self.action["treatment"] = request_countl.random()
# Then, give the condition with the lowest fulfilled surveys
else:
    # We select the action with the least amount of fulfilled
    ↪  surveys
    self.action['treatment'] = fulfill_countl.min()

# Increase request count before storing it in the database again
count = base.Count(self.get_theta(key="request_count",
↪  value=self.action["treatment"]))
# Again we can use a built-in function to increment the count
count.increment()
self.set_theta(count, key="request_count",
↪  value=self.action["treatment"])
```

(a) Code for selecting an action in a between-subjects design with balancing.

```
# Retrieve the count of fulfilled surveys from the database
count = base.Count(self.get_theta(key="fulfilled_surveys",
↪  value=self.action["treatment"]))
# Update the count using a built-in update function
count.update(self.reward["finished"])
# Store the count in the database
self.set_theta(count, key="fulfilled_surveys",
↪  value=self.action["treatment"])
```

(b) Code for setting a reward in a between-subjects design with balancing.



Then Branch If:

If action.treatment Is Equal to 1

Show Block: Group 1 (1 Question)

Web Service

URL: http://HOST:8080/setreward/3c9d80a4a0

Method: GET

Query Parameters

key = 3bcde858f6

action = {"treatment":1}

reward = {"finished":1}

□ Fire and Forget

Set Embedded Data

End of Survey

(c) An extended branch in the *Survey Flow* that returns a reward when the survey is finished, to keep count of the number of finished surveys.

Figure 5.5: The flow for a between-subjects design with balancing of conditions implemented in **Qualtrics** using **StreamingBandit**.

And with a few adaptations, we have made sure that the conditions will be proportionally balanced. The subjects will not experience a different survey than in the regular between-subjects design, but the allocation of the treatments will be different, as shown in Table 5.3.

Table 5.3: An example of the treatment allocation for a between-subjects design with balancing. After 1000 interactions, the treatment with the lowest amount of fulfilled surveys is selected – in this case it is condition 2 – which ensures that the number of 1's and 2's is (almost) always equal.

Interaction	Selected treatment
1	1
2	2
3	2
4	1
⋮	⋮
1001	2
1002	2

5.4.3 Within-subjects and mixed designs

In some experiments the design requires the subjects to take tests multiple times – either under different conditions or over multiple time periods. This type of design (a within-subjects or repeated measures design) requires the experimenter to be able to deal with returning subjects as they will have to take test on different moments. This means that we have to remember a) who the subject is and b) which tests the subject took up until the current moment. Using **StreamingBandit** we can implement these requirements from within the getAction call: we will first check if a new identifier needs to be created (i.e. is this subject returning?) and we can then give a random action based on the previous tests of the user. Note that with this implementation we are not only able to implement a within-subjects design, but can also be used to set up a mixed design for example (a combination of a within-subjects and between-subjects design). The specification of the treatment conditions can be a combination of multiple independent variables (as in mixed designs). In this example, we will use a within-subject design with three different treatment conditions: "A", "B", "C". Figure 5.6 shows the complete steps.

1. (see Figure 5.6a) First, in **Qualtrics** we will have a first block that asks for a user identifier, and if it is a new subject it asks the subject to click *Next* to get a new user identifier. Entering the user identifier is done using a text entry question in **Qualtrics**.

2. (see Figure 5.6b) Then we make sure the given result (the user identifier) is saved in an *Embedded Data* field, so that we can use the response to send to **StreamingBandit**.

3. (see Figure 5.6c) We can now send a call to **StreamingBandit**, but we have not implemented any allocation procedure in **StreamingBandit**. The block in Figure 5.6c shows the code to get the desired treatment allocation behavior in the getAction call. First of all, if the `user_id` field is empty (i.e. a subject pushed the next button and is a new subject), we will generate a new random string.[4] Then we generate a random action. If the `user_id` field is not empty, we check if it is a correct identifier by checking if the returned result from the database is not empty (i.e. the user has already finished a test before). If this is the case, we return another random treatment or if the subject has already done all three tests we return a "finished". If the subject has input a wrong identifier, we return "wrong_id". Both of these have to be handled within **Qualtrics** in the *Survey Flow* (see Figure 5.6g).

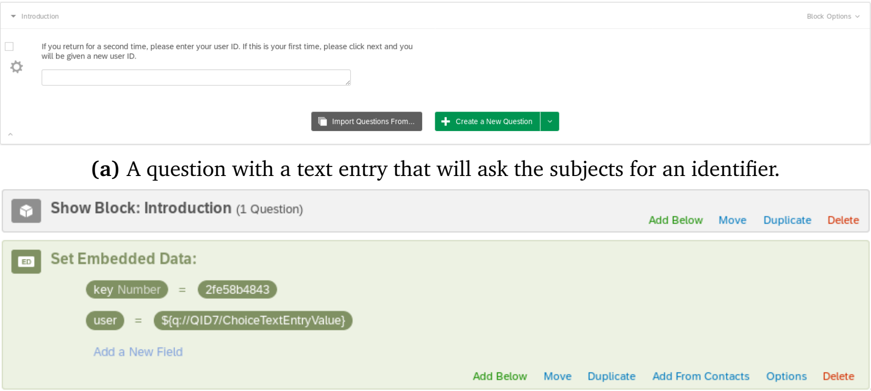
[4] To make sure it is not a duplicate (which is highly unlikely) we check it against the identifiers of all other subjects (in the `while`-loop).

4. (see Figure 5.6d) Then, we use the code for the setReward call as shown in Figure 5.6d. Here we update the count of the finished treatments so that next time we will know which condition(s) the subject already has been through.

5. (see Figure 5.6e and 5.6f) Now we can set up the calls within **Qualtrics**. We do this using two branches, where in the first branch the subject is a new subject (i.e. the text entry was left empty), this is done in Figure 5.6e. We set up the *Web Service* similar to the ones we have done before, but now also make sure to save the returned identifier in an *Embedded Data* field. Then we can show a block with the new identifier to the subject to save for later. Figure 5.6f shows the text block with the content that will be shown to a new user.

6. (see Figure 5.6g and 5.6h) The second branch is shown in Figure 5.6g. If the subject has input an identifier (i.e. the text entry was not left empty), we can send this to **StreamingBandit**. In the returned `action.treatment`, we can now see whether the input identifier was correct. If this was incorrect, we send the user to the end of the survey. Figure 5.6h shows the text block with content that will be shown to a returning user.

7. (see Figure 5.6i and 5.6j) Figure 5.6i then shows how to branch for the different treatment conditions. "A" is shown here, but it would be similar for the other conditions as well. Figure 5.6j shows a branch for when the subject has finished all the tests (i.e. the returned `action.treatment` is equal to "finished").



If you return for a second time, please enter your user ID. If this is your first time, please click next and you will be given a new user ID.

(a) A question with a text entry that will ask the subjects for an identifier.

key Number = 2fe58b4843

user = ${q://QID7/ChoiceTextEntryValue}

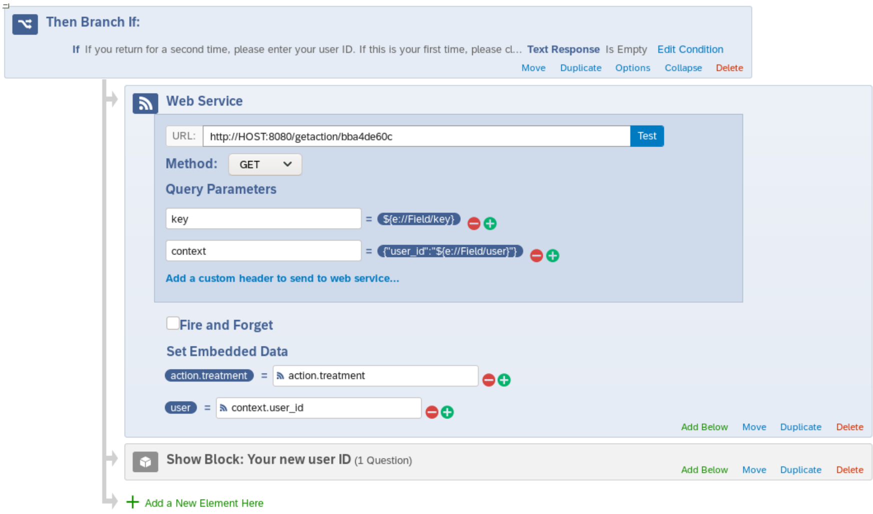
(b) Saving the identifier in an *Embedded Data* field in the *Survey Flow* of **Qualtrics**.

```
# If we have a new user, we generate a new random string
if self.context["user_id"] == "":
    # Generate a random string
    self.context["user_id"] = hex(random.getrandbits(42))[2:-1]

    # If the string already exists in the database, generate a
    ↪  new one until it is unique
    while self.get_theta(key="finished" +
    ↪  self.context["user_id"]) != {}:
        self.context["user_id"] =
        ↪  hex(random.getrandbits(42))[2:-1]

    # Generate a random action using the base.List class and
    ↪  the random() function
    member_finished = base.List(self.get_theta(key="finished" +
    ↪  self.context["user_id"]), base.Count, ["A", "B", "C"])
    self.action["treatment"] = member_finished.random()
# or if we have an existing user, we generate an action that
↪  has not been shown yet
elif self.get_theta(key="finished" + self.context["user_id"])
↪  != {}:
    # Retrieve the finished conditions from the database
    member_finished = base.List(self.get_theta(key="finished" +
    ↪  self.context["user_id"]), base.Count, ["A", "B", "C"])
    # If the user still has not done all three conditions,
    ↪  select a random one
    if member_finished.count() < 3:
        self.action["treatment"] = member_finished.random()
        # If we selected a random action, check if it has been
        ↪  played already
        while int(member_finished.get_dict()[self.action["treat⌋
        ↪  ment"]]['n']) >
        ↪  0:
            # If so, generate a new action until we find one
            ↪  that has not been played yet
            self.action["treatment"] = member_finished.random()
    # If the user has done all conditions, return a "finished"
    else:
        self.action["treatment"] = "finished"
# Assertion: if the user has supplied a wrong identifier
else:
    self.action["treatment"] = "wrong_id"
```

(c) Code for selecting an action in a within-subjects or mixed design.

```
# If a condition was finished, we can update the parameters in
↪  the database
if self.reward["finished"] is 1:
    # Retrieve the counts from the database for the played
    ↪  condition
    finished_treatment =
    ↪  base.Count(self.get_theta(key="finished" +
    ↪  self.context["user_id"],
    ↪  value=self.action["treatment"]))
    # Increment the count
    finished_treatment.increment()
    # Store it in the database again
    self.set_theta(finished_treatment, key="finished" +
    ↪  self.context["user_id"], value=self.action["treatment"])
```

(d) Code for setting a reward in a witin-subjects and mixed design.



Then Branch If:

If If you return for a second time, please enter your user ID. If this is your first time, please cl... **Text Response** Is Empty

Web Service

URL: http://HOST:8080/getaction/bba4de60c

Method: GET

Query Parameters

key = ${e://Field/key}

context = {"user_id":"${e://Field/user}"}

□ Fire and Forget

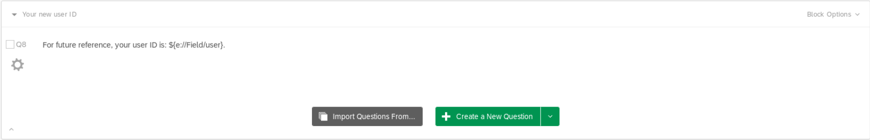
Set Embedded Data

action.treatment = action.treatment

user = context.user_id

Show Block: Your new user ID (1 Question)

(e) A branch in the *Survey Flow* that handles a new subject.

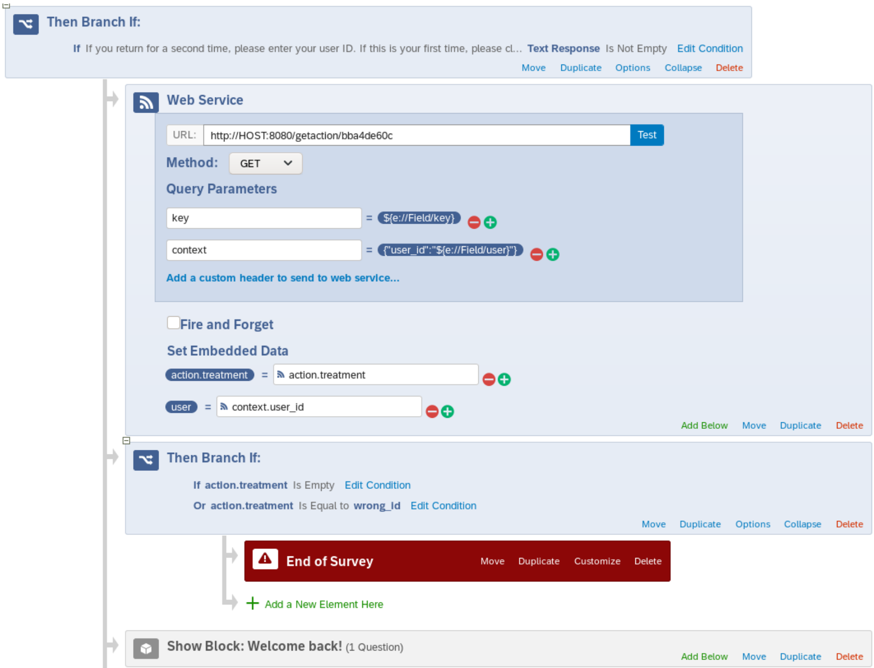


Your new user ID

Q8 For future reference, your user ID is: ${e://Field/user}.

(f) The text block with the content that will be shown to new subjects.

5



Then Branch If:

If If you return for a second time, please enter your user ID. If this is your first time, please cl... **Text Response** Is Not Empty

Web Service

URL: http://HOST:8080/getaction/bba4de60c

Method: GET

Query Parameters

key	=	${e://Field/key}
context	=	{"user_id":"${e://Field/user}"}

Add a custom header to send to web service...

□ Fire and Forget

Set Embedded Data

action.treatment	=	action.treatment
user	=	context.user_id

Then Branch If:

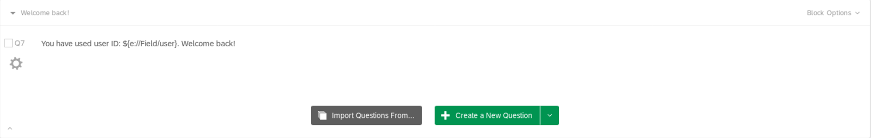
If **action.treatment** Is Empty

Or **action.treatment** Is Equal to **wrong_id**

End of Survey

Show Block: Welcome back! (1 Question)

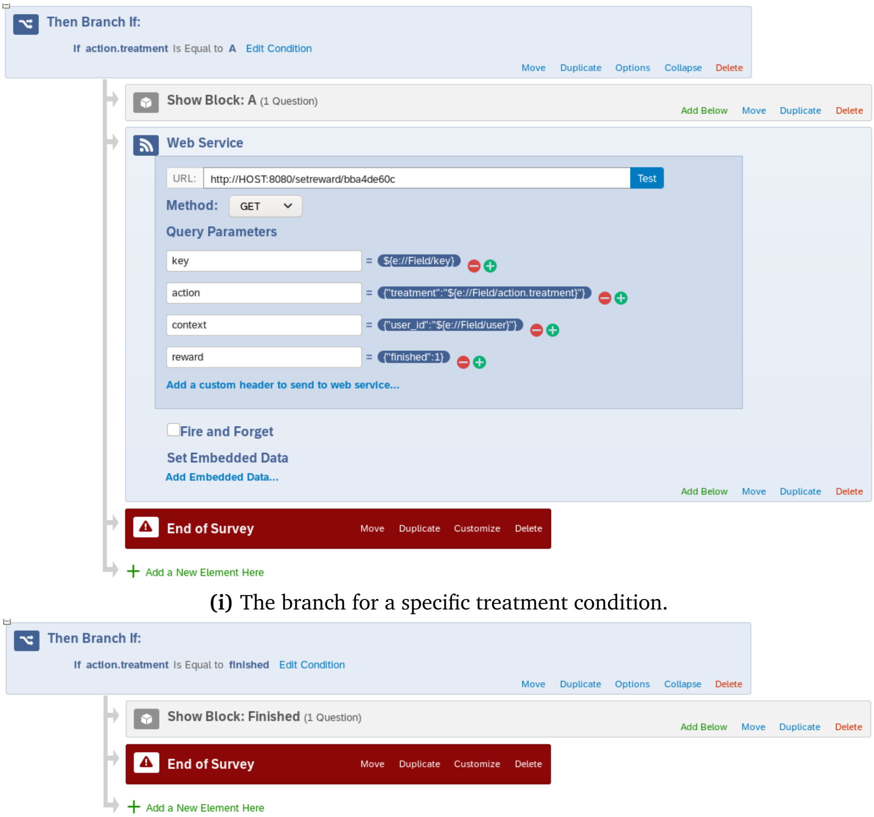
(g) A branch in the *Survey Flow* that handles a returning subject.



Welcome back!

Q7 You have used user ID: ${e://Field/user}. Welcome back!

(h) The text block with the content that will be shown to returning subjects.



Then Branch If:

If action.treatment Is Equal to A

Show Block: A (1 Question)

Web Service

URL: http://HOST:8080/setreward/bba4de60c

Method: GET

Query Parameters

key = ${e://Field/key}

action = {"treatment":"${e://Field/action.treatment}"}

context = {"user_id":"${e://Field/user}"}

reward = {"finished":1}

□ Fire and Forget

Set Embedded Data

End of Survey

(i) The branch for a specific treatment condition.

Then Branch If:

If action.treatment Is Equal to finished

Show Block: Finished (1 Question)

End of Survey

(j) The branch for when the subject has finished all the tests.

Figure 5.6: The flow for a within-subjects or mixed design implemented in **Qualtrics** using **StreamingBandit**.

Figure 5.7 shows an overview of how the subject will arrive at the different survey conditions. This feature of remembering a subject identifier is very useful and the use is not limited to just doing a within-subjects design. For example, it could also be used when the experimenter wants to the subject to be able to save the survey and complete it at a later time.

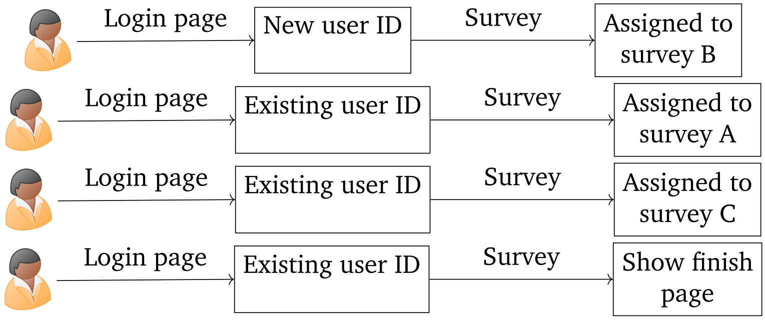


Figure 5.7: Visual overview of how one and the same subject experiences the within-subjects survey. The first time, the subject will be supplied with a user identifier and assigned to a random condition. Then after supplying the user identifier two more times the subject will be assigned to a random condition that it has not experienced yet. Finally, the subject will be redirected to a finished page when it tries to login again.

5.4.4 Thompson sampling for increasing estimation precision

The previous examples display the implementation details of relatively common experimental designs. Even there, **StreamingBandit** allows us to advance the design of experiments in **Qualtrics**. In this example we illustrate how to take an extra step by implementing an adaptive design. In this specific case, we are interested in making sure that the we will have precise estimates of our effects in an experiment where we (a priori) assume that the variances of the observations differs between two experimental conditions. When the variances of the observations of the treatment conditions are assumed to be unequal, it pays off to increase the number of subjects allocated to the highest variance condition to ultimately improve the precision of the mean difference estimator (Kaptein, 2014).

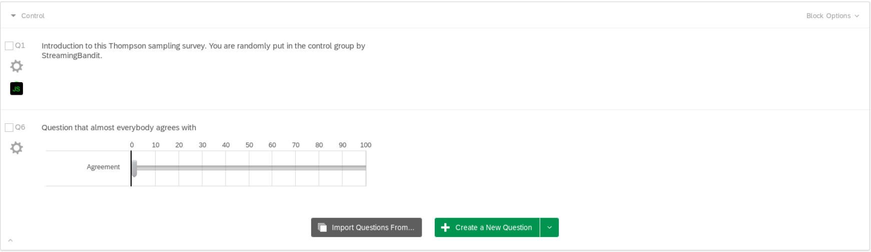
For example, assume we have a survey with two groups: one group receives questions with statements that people tend to agree on (i.e. we assume a low variance in the observations) and another group receives questions with statements that people do not tend to agree on (i.e. we assume a high variance). Kaptein (2014) uses Thompson sampling and computes the posterior variance for each condition (i.e. σ_1^2 and σ_2^2) using a normal-inverse χ^2 model (to be done in the setReward call). Then in the getAction call a draw d from each of the two posteriors is obtained and the treatment is selected for which $\frac{d}{n}$ is highest, where n is the number of subjects that were previously allocated to that treatment. Thus, the condition with the highest expected variance will be selected. Another benefit of this is that if the prior assumption of unequal variances does not hold true, this sampling scheme will still balance the number of subjects accordingly. This sampling scheme has already been implemented in **StreamingBandit**, so our code will be relatively easy to generate, but we will explain the steps. Figure 5.8 shows the complete steps of implementing a Thompson sampling scheme for increasing estimation precision in a scenario of questions with potentially different outcomes on the variance.

1. (see Figure 5.8a) First we set up the different conditions with different questions. Figure 5.8a shows an example for our first condition (i.e. assumed low variance). We set up a question that “almost everybody agrees with” and use a slider to give an agreement from 0 to 100. We do this exactly the same for the second condition, but we simply change the question into a question that “people tend not to agree on”.

2. (see Figure 5.8b) Then for the code: in the getAction code we retreive the data from the database and using the built-in `experimentThompson` function we select the treatment condition.

3. (see Figure 5.8c) When a survey has been completed, we can update the variance of the selected condition in the setReward call, which can be any continuous reward. Integrating these calls into **Qualtrics** will be similar to the previous designs, so for an example look at Figure 5.2b.

4. (see Figure 5.8d) Now in returning the reward, we have to specify what the returned reward will be. In **Qualtrics** we can select this reward to be an *Embedded Data* field, which is the answer to our questions. We can use the total reward of that question. Of course, we can use different measures here, another possible option would be to return the mean of different ratings (e.g. a likert scale rating).



Q1 Introduction to this Thompson sampling survey. You are randomly put in the control group by StreamingBandit.

Q6 Question that almost everybody agrees with

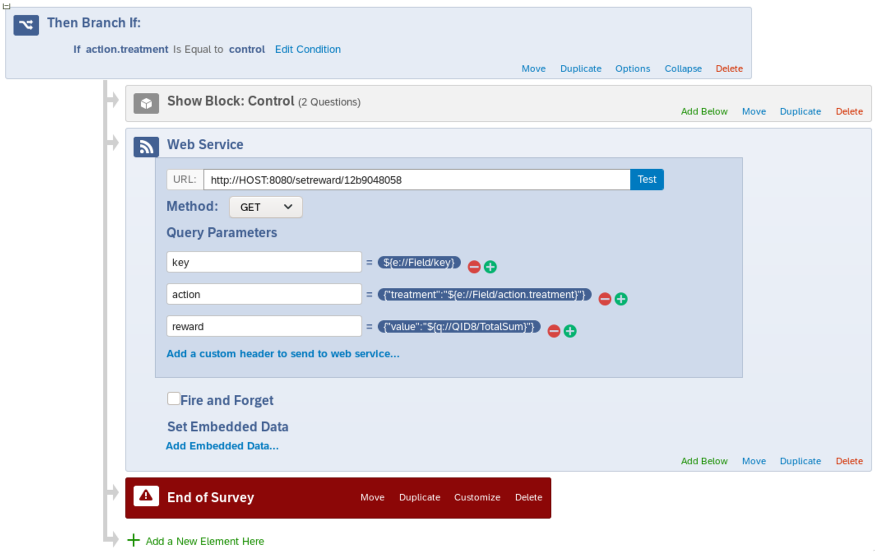
(a) Example of a question block with a question with a continuous outcome.

```
# Retrieve the parameters from the database and store it in the
↪   built-in ThompsonVarList class
var_list =
↪   thmp.ThompsonVarList(self.get_theta(key="treatment"),
↪   ["control","treatment"])
# Select an action using the built-in sampling function
self.action["treatment"] = var_list.experimentThompson()
```

(b) Code for selecting an action using a Thompson sampling scheme for increasing estimation precision.

```
# Retrieve the variance parameters for the played action
var = base.Variance(self.get_theta(key="treatment",
↪   value=self.action["treatment"]))
# Update the variance using a built-in update function
var.update(self.reward["value"])
# Store the parameters in the database
self.set_theta(var, key="treatment",
↪   value=self.action["treatment"])
```

(c) Code for setting a reward for the Thompson sampling scheme for increasing estimation precision.



If action.treatment Is Equal to control

URL: http://HOST:8080/setreward/12b9048058

Method: GET

key = ${e://Field/key}

action = {"treatment":"${e://Field/action.treatment}"}

reward = {"value":"${q://QID8/TotalSum}"}

□ Fire and Forget

(d) A branch in the *Survey Flow* for our Thompson sampling scheme, which is very similar to previous branches, with the difference being that we return a total sum of our questions in the reward.

Figure 5.8: The flow for our Thompson sampling scheme implemented in **Qualtrics** using **StreamingBandit**.

Table 5.4: An example of the treatment allocation for a Thompson sampling scheme for increasing estimation precision. For the first few subjects, the precision of the estimates for both conditions are relatively even, which results in randomly selectting either of the two conditions. After 50 subjects, the treatment condition (i.e. the condition with the highest observed variance) will be selected more often until the precision has increased. After 100 subjects, the precision is relatively even again.

Interaction	Selected treatment	Condition with lowest expected precision
1	control	approximately equal
2	treatment	approximately equal
3	treatment	approximately equal
4	control	approximately equal
⋮	⋮	⋮
50	treatment	treatment
51	treatment	treatment
⋮	⋮	⋮
100	control	approximately equal
101	treatment	approximately equal

This setup now ensures that we balance the different conditions based on the variance of the observed outcome variable. Table 5.4 shows an example of how the treatments can be allocated. As subjects come in, this adaptive design will select the treatment with the highest observed variance, which in turn ensures that the eventual estimation of the effect will be approximately equally precise for both conditions.

5.5 Recent field example

To demonstrate an implementation of a sequential allocation procedure in practice, we will take a look at the experiment done by Kaptein, van Emden, and Iannuzzi (2016b). The authors wanted to re-affirm the existence of the heavily debated decoy effect using a sequential allocation procedure. The decoy effect describes a human bias when deciding between three offers, where one of the offers is a decoy that rationally should have no influence on the decision (see e.g., Huber, Payne, & Puto, 1982). What is found, however, is that this decoy does have influence on the decision making process as people tend to go for the more expensive offer in the presence of a decoy. One example of the decoy effect is in a scenario where the subject wants to buy a laptop, as shown in Figure 5.9. In the left example three options are presented: the option with 1.1 GB RAM is a so-called decoy to attract more potential buyers to the option with 2.0 GB RAM,

You need a new laptop computer. All other things being equal about the options, which laptop would you choose?	You need a new laptop computer. All other things being equal about the options, which laptop would you choose?
o 4.0 GB RAM - 14 Hrs Average Battery Life	o 4.0 GB RAM - 14 Hrs Average Battery Life
o 1.1 GB RAM - 20 Hrs Average Battery Life	o 2.0 GB RAM - 24 Hrs Average Battery Life
o 2.0 GB RAM - 24 Hrs Average Battery Life	

Figure 5.9: An example of the laptop scenario used in the decoy effect study. Left is shown an example with a decoy, right is an example without a decoy.

which is a competitor for the 4.0 GB RAM. The example on the right is a scenario without a decoy.

The decoy effect has been heavily debated in research, where some replications of earlier research disputed the practical relevance and existence of the effect. In their paper Kaptein, van Emden, and Iannuzzi (2016b) argue that this is likely due to the suboptimal placings of the decoys in a grid of properties (i.e. the values that the decoy can take are suboptimal) and use a novel sequential allocation procedure to automatically find the decoy that maximizes the choice switch. Designing the experiment using a sequential allocation procedure allowed the research to be effective where it otherwise could not: earlier research was hampered by the fact that the decoy had to be determined a priori, which could result in not finding an effect or experiments that could not be reproduced.

The sequential allocation procedure used in the experiment is called Lock-in Feedback (LiF). LiF is a policy developed for the continuous-armed bandit problem (CAB). In the CAB problem, the actions are based on a continuous range instead of a discrete number of arms – for example choosing a price from a price range for a product. LiF aims to seek the maximum of a continuous function $y = f(x)$ (or a reward function), where y in this case would be the probability of switching the choice and x would be an attribute that you wish to change (i.e. the battery life in a laptop). The maximization of the function is done using oscillation on the attribute x with time, these oscillations introduce variance in the chosen attributes. Using rewards (feedback) on these chosen attributes, LiF tries to find the optimum of y. For more details on LiF and how it works, see Kaptein and Ianuzzi (2016).

The experiment was done using a customized front-end survey that was written in **Drupal** (a PHP based framework to build websites) (Buytaert et al., 2001). The subjects were recruited using Amazon's **MTurk** (Amazon, 2012). The authors considered five different scenarios to use the decoy, of which the laptop example was one. If the scenarios had multiple attributes (e.g. a laptop scenario with different size of RAM and battery life), they would only change one attribute dimension (e.g. only the battery life) for the decoy. The experiment used a between-subjects design, where one of the three conditions was a sequential allocation procedure.

These three conditions were:

1. Baseline: no decoy was present and participants were presented with a choice between two products.
2. Random: a decoy was present, but the different values for the decoy were selected based on hardcoded values tailored to the specific scenario at play. The hardcoded values were supplied in the `context` of the getAction call (i.e. `self.context["min"]` and `self.context["max"]`).
3. Lock-in Feedback: a decoy was present and the values of the decoy were selected based on previous interactions in the experiment and the parameters of the LiF policy. The LiF policy was updated based on the selected item by the participants. As LiF contains a multitude of tuning parameters, we refer the readers to Kaptein, van Emden, and Iannuzzi (2016b) for more details.

The flexibility of **StreamingBandit** allowed the researchers to completely implement this complex sampling scheme relatively easy. The code for the getAction call looks as follows:

```
# If we have not assigned a condition to a user yet (i.e. new
↪  user), do so
if not("condition" in self.get_theta("user_id",
↪  self.context["user_id"])):
    # Note that this is the first allocation to the Drupal survey
    self.action["note"] = "First allocation"
    # Randomly select an action
    draw = random.choice(["baseline", "random", "lockin"])
    # Store the condition in the database
    self.set_theta({"condition":draw}, "userid",
    ↪  self.context["userid"])
# Set the action from the selected condition (also for the next
↪  time a user comes in)
self.action["condition"] = self.get_theta("userid",
↪  self.context["userid"])["condition"]

# If the condition is baseline, we do not need to select a decoy
if self.action["condition"] == "baseline":
    # And we can return "none"
    self.action["decoy"] = "none"
# If the condition is random, we select a random decoy
```

```
elif self.action["condition"] == "random":
    # This is done using the supplied bounds (min and max in the
    ↪   context)
    self.action["decoy"] = np.random.uniform(self.context["min"],
    ↪   self.context["max"])
# If the condition is lockin, we use the LiF policy to select a
↪   decoy
elif self.action["condition"] == "lockin":
    # We retrieve the parameters from the database
    theta = self.get_theta(all_float=False, key="question",
    ↪   value=self.context["question"])
    # Using the supplied parameters, we instantiate a LiF object
    lif_pol = lif.LiF(theta, x0=self.context['x0'],
    ↪   A=self.context['A'], T=150, gamma=.06, omega=1.0,
    ↪   lifversion=1)
    # Using the built-in function, we select ("suggest") an action
    suggestion = lif_pol.suggest()
    # We make sure that the suggestion will not be lower than 0
    if suggestion["x"] < 0:
        self.action["x"] = 0
    else:
        self.action["x"] = suggestion["x"]
    # We also supply the t and x0, so that LiF can be updated the
    ↪   next time again
    self.action["t"] = suggestion["t"]
    self.action["x0"] = suggestion["x0"]
```

As the experiment progresses through the five different scenarios, we have to remember which condition a participant was put in. The **Drupal** survey supplied the user indentifier in the context. The first lines make sure that if the user identifier is not known, they get assigned a condition. Then the condition is retrieved from the database and based on the condition, three different things can happen. Either we return no decoy (first condition), we return a random decoy based on the supplied hardcoded values (second condition) or (third condition) we retrieve the parameters for LiF from the database (and the context supplies which scenario (or question) is being shown now) and select a suggested action from LiF.

Then to update the parameters, we have the following code for the setReward call:

```
# We only need to update parameters when LiF is the condition
if self.action["condition"] == "lockin":
    # Get the parameters from the database
    theta = self.get_theta(all_float=False, key="question",
    ↪   value=self.context["question"])
    # Instantiate a LiF class again
    lif_pol = lif.LiF(theta, x0=self.context['x0'],
    ↪   A=self.context['A'], T=150, gamma=.06, omega=1.0,
    ↪   lifversion=1)
    # Update the parameters using a built-in update function
    lif_pol.update(self.action["t"], self.action["x"],
    ↪   self.reward["r"], self.action["x0"])
    # Store the parameters in the database
    self.set_theta(lif_pol, key="question",
    ↪   value=self.context["question"])
```

We only have to update parameters if the selected condition was LiF (checked using the `if`-statement). Here we do the same thing as in other examples: we get the parameters from the database (only the parameters for the specific question, given in the context), update the instantiated LiF object and store it in the database again. In their experiment, the authors have shown that a) the decoy effect does definitely exist and b) that sequential allocation procedures are a viable option of finding the most optimal position for the decoy – a conclusion that was hard to draw with other experimental designs, and **StreamingBandit** played a crucial role.

5.6 Conclusions

In this paper we have shown how experimenters can effectively design experiments that use complex, sequential treatment allocation schemes in web-based (front-end) platforms using **StreamingBandit** as a back-end. We provided an extensive tutorial on how to implement a variety of treatment allocation schemes that can be consumed by a front-end platform. The presented work provides a generic framework that allows researchers to easily adapt and test any type of treatment allocation procedure. As shown, for example, going from a between-subjects design to a Thompson sampling policy only requires changing a few lines of code in **StreamingBandit**.

One limitation of **StreamingBandit** is that it requires the desired front-end platform to be capable of integrating API requests. This is a very common internet standard, however, which was one of the design considerations for **StreamingBandit**. Nevertheless, once **StreamingBandit** is integrated it provides an extremely flexible platform for the design of experiments.

StreamingBandit serves to fulfill the opportunity to use sequential decision problems in behavioral sciences identified in Eckles and Kaptein (2019). It has already been applied in multiple experiments, such as the example of the decoy experiment. Other examples of applying **StreamingBandit** in practice are also shown in Kaptein et al. (2017), Kruijswijk, Parvinen, and Kaptein (2019) and Parvin, Chessa, Kaptein, and Paternò (2019) and these examples also display a hint of cases in which sequential allocation procedures might be beneficial to apply. We hope that this paper will enable the adaptation of sequential allocation procedures in future (web-based) research.

Chapter 6

Epilogue

6.1 Overview

With this thesis we aimed to introduce, and extend the use of, sequential allocation procedures to researchers from social sciences and other applied fields. Another goal was to further develop the methodology of the multi-armed bandit problem and develop a flexible software tool to be able to easily implement sequential allocation procedures in experiments. We provided various empirical examples, illustrating the usefulness and applicability of experiments using sequential allocation procedures: in Chapter 2 and 5 we showed how to set up an experiment with a sequential allocation procedure for a decoy experiment, in Chapter 2 and 4 how to collect and analyze field data, and in Chapter 3 another offline analysis of existing empirical data. To further facilitate the use of the developed tools, all the work in this thesis is open source. As a result, **StreamingBandit** is actively being picked up, used, and improved by the open-source community.

In Chapter 2 and 5, we introduced **StreamingBandit** as a framework and software application. The framework identifies how multi-armed bandit policies can be summarized into two steps: a *summary* step (where we update the parameters of a policy) and a *decision* step (where we use the policy to choose an action). We introduced the software and next to a detailed explanation, we provided ample examples of how **StreamingBandit** can be utilized to implement (sequential) allocation procedures in (web-based) research. We demonstrated its flexibility by showing how the software can be used to implement any type of allocation procedure, whether it is sequential in nature or not. **StreamingBandit** has already shown its capabilities in several experiments, some of which have been highlighted throughout the chapters (Kaptein et al., 2017; Kaptein, van Emden, & Iannuzzi, 2016b; Kruijswijk et al., 2019; Parvin et al., 2019).

In Chapter 3, we introduced methods to model hierarchical dependencies in several types of multi-armed bandit policies. We showed, through simulations and an empirical study, that taking hierarchical dependencies into account in the policies improves the performance when such dependencies are – potentially, in the case of empirical studies – present. We used methods that are commonly known in the more traditional statistical literature: a shrinkage factor approach (Ippel et al., 2019) and a Bayesian hierarchical modeling approach, which both showed improvements when incorporated in the existing MAB policies.

In Chapter 4, we extended an offline evaluation method developed by Li et al. (2011) such that it can also be used for continuous-armed bandit problems. Furthermore, we compared our method to the work of Kallus and Zhou (2018), who introduced an offline evaluation method for static policies with continuous treatments – this work was introduced in the literature after we first started working on our method. Our work showed a relatively consistent ranking of policies in offline evaluations. Furthermore, we showed how to use both our method and the method by Kallus and Zhou (2018) for offline parameter tuning of continuous-armed bandit policies.

6.2 Further methodological considerations

In this thesis, we mainly discussed our methods and tools in light of solving the goal of the canonical multi-armed bandit problem: through sequential interactions, try to maximize the outcome over all interactions. Although such a goal serves its purpose for many social science applications (as shown throughout the chapters), it is not suitable for every type of experiment. For example, contrary to maximizing the overall reward, in the traditional experiment we often focus on estimating the effects of the treatments under consideration, or on selecting the overall best treatment. In this section, we briefly touch upon considerations that should be taken when choosing for sequential allocation procedures in such cases and we highlight alternative uses of sequential allocation procedures (some of which were already briefly mentioned in the previous chapters). These methodological considerations are:

- *Hypothesis testing*: testing hypotheses in experimental comparisons (e.g. difference in means between two groups),
- *Best-arm identification*: select the best performing treatment within a given number of possible interactions,
- *Optimal design*: designing an experiment to sample as optimally as possible with respect to a statistical estimate of interest.

Rather than discussing these alternative goals and methodologies in full, we suggest the reader to consult the cited sources for in depth discussions on these topics. However, we would like to stress that these alternative methodologies can, and often very easily, also be implemented using **StreamingBandit**.

6.2.1 Hypothesis testing

The most common use of randomized controlled trials is for testing hypotheses on differences between experimental conditions. When using sequential allocation procedures, one of the biggest cautions that should be taken is to avoid using improper methods of hypothesis testing – for example when the methods are designed to deal with fixed sample sizes, which is violated in the sequential treatment allocation case. While typically the necessary sample size to achieve the required power of the relevant tests is determined a priori, it is also often the case that due to (e.g.) costs, an as small as possible number of subjects is preferred. However, stopping data collection based on continuous updating of the p-value for a specified null hypothesis, as appealing as this might look, can and will often lead to inflated type I error rates, and thus to invalid conclusions (this is also known as the multiple testing problem (Hsu, 1996)). To tackle this problem, solutions have been proposed for obtaining so-called always valid p-values (Johari, Pekelis, & Walsh, 2015) and controlling (online) false discovery rates (Jamieson & Jain, 2018; Yang, Ramdas, Jamieson, & Wainwright, 2017).

Closely related to the literature on hypothesis testing is the vast literature on Bayesian adaptive clinical trials (see e.g., Berry, 2006, 2012). In this literature, several aspects of the design of experiments are studied, among which is early stopping. Early stopping is often applied in high-cost experiments (e.g., in clinical trials) which should not be run longer than necessary. Using sequential experimentation and Bayesian statistics allows stopping rules to be used to adapt the experiment based on the observed data as it is coming in, instead of, for example, relying on the a priori sample size computations (which often rely on possible wrong values for the effect sizes).

6.2.2 Best-arm identification

One interesting alternative to the MAB problem is that of best-arm identification (also called the pure exploration problem) (Audibert, Bubeck, & Munos, 2010): given a fixed number of possible interactions and a set of treatments, select the best performing treatment. The goal now is not to, over time, maximize the cumulative reward (or minimize the regret), but to maximize the probability of selecting the best treatment given those fixed number of interactions. In this scenario, we are interested in distinguishing the single best treatment between the best possible treatments, and we thus want to be as confident as possible about the choosing the best arm given those fixed number of interactions. We therefore want to explore more between multiple promising candidates. This is in contrast to the traditional MAB problem, where we are interested in losing as little reward as possible during exploration (Kaufmann, Cappé, & Garivier, 2016). See, for example, Jamieson and Nowak (2014), Russo (2016) for simple policies for best arm identification and Libin et al. (2018) for an application in an epidemiological setting.

6.2.3 Optimal design

The literature that concerns the optimal design of experiments deals with methods for developing study designs that are optimal with respect to statistical estimates of interest – for example, minimizing the standard errors of the effect size estimates (Chernoff, 1972; Goos & Jones, 2011). In such a scenario, the view of sequentially allocation treatments can be beneficial, although traditionally the optimal design literature is not focused on sequential allocation. One example that has already been highlighted in Chapter 2 and 5 is the use of a Thompson sampling scheme to increase estimation precision when estimating the difference between two means for two groups with unequal variances (Kaptein, 2014). In the traditional setting, we rely again on a priori assumptions that may not be true (or less valid) in practice. By using data collected during the experiment, we can adapt the experiment while running and improve estimation precision or efficiency even more (Kaptein, 2014; Ryan, Drovandi, McGree, & Pettitt, 2016; Whitehead & Brunier, 1995).

6.3 Future research directions

Next to looking at other methodological alternatives as we did in the previous section, several directions for future research can be identified for each chapter based on the results of this thesis, which will contribute to the growth of the methodology for, and use of, sequential allocation procedures.

Further work could be done to improve the outreach and the user-friendliness of **StreamingBandit**, introduced in Chapter 2. Firstly, the default policies and libraries could be improved and expanded to contain a more diverse set of sequential allocation procedures. Although **StreamingBandit** allows researchers to implement any type of allocation procedure, having a larger set of default options would allow researchers who are less experienced with Python and such tools in general to make more optimal use of the strength of **StreamingBandit**. Secondly, the process of installing and maintaining **StreamingBandit** could be improved to increase the outreach of the software. In that sense, the installation of **StreamingBandit** is relatively straightforward provided that there is experience with either **Docker** (Merkel, 2014) or handling the command-line. Providing **StreamingBandit** as a binary installation format or as *Software as a Service* (SaaS), where users can directly use a live version of **StreamingBandit** hosted on a cloud platform, would make using the software even easier. However, running a SaaS application brings in a factor of maintenance costs and time, which makes it harder to implement.

The work in Chapter 3 could be extended in multiple ways. Firstly, as identified in the discussion of the chapter, further research could be done in identifying useful methods for partial pooling, such as more efficient ways of estimating posteriors for the Bayesian models. While hierarchical models have proven useful throughout the social sciences, they are often hard to update and evaluate online – a feature that is necessary for most practical CMAB applications (Agarwal et al., 2017; Ansari & Mela, 2003; Ippel et al., 2016b). For the Bayesian models, this could for example be done via sequential Monte Carlo methods (Doucet et al., 2001) or bootstrapped Thompson sampling (Eckles & Kaptein, 2019). Secondly, as a large portion of the bandit literature focusses on the contextual MAB problem, the methods discussed in this chapter could be used to improve state-of-the-art contextual MAB policies, such as the popular LinUCB algorithm (Li et al., 2010).

We developed the delta method in Chapter 4 as a simple way of evaluating continuous-armed bandit policies offline. Our solution provides initial steps towards effective offline policy evaluation for dynamic policies. Although our current solution showed relative consistent ranking, there is still room for improvement. Two obvious potential options could be: 1) extend the Doubly Robust method developed in Dudík et al. (2014) using regression methods and inverse propensity scoring for continuous treatments (Hirano & Imbens, 2004) and 2) try to combine the kernel method of Kallus and Zhou (2018) with our evaluation method – for example to use kernel smoothing on the rewards when the proposed actions fall within the acceptance range.

While thesis is more practically oriented than theory driven, especially the topics discussed in Chapter 3 and 4 could be put under more theoretical scrutiny. For Chapter 3 regret bounds could have been derived depending on specific environmental assumptions (e.g., the distribution of the rewards), which becomes even more important with complex (contextual) MAB policies using shrinkage factors or Bayesian hierarchical modeling – such as an extension of the LinUCB algorithm. For Chapter 4 the extra theoretical scrutiny includes further scrutinizing the effect of the tuning parameter δ and performing additional bias analyses for possible extensions of the proposed delta method. Nevertheless, we believe that simulation and empirical (offline) evaluation studies are complementary to theoretical work, as a) they typically rely on less (or weaker) assumptions, b) many algorithms tend to be impractical to use, and c) their empirical behavior is often poorly understood, as also identified in Bietti, Agarwal, and Langford (2018).

Finally, more work could be done on enabling less technical researchers to incorporate complex policies in their research. Future research could be done on determining which (type of) sequential allocation procedure is useful in which (type of) experiment. As an example, **contextual** is a useful R package that provides tools to simulate and develop policies for different multi-armed bandit problems (van Emden & Kaptein, 2018). Such work is important to further expedite the use of sequential allocation procedures in experiments, especially for applied researchers.

Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., . . . Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In K. Keeton & T. Roscoe (Eds.), *12th USENIX symposium on operating systems design and implementation, osdi* (pp. 265–283). USENIX Association.

Agarwal, A., Bird, S., Cozowicz, M., Hoang, L., Langford, J., Lee, S., . . . Slivkins, A. (2017). Making contextual decisions with low technical debt. eprint: arXiv: 1606.03966

Agarwal, A., Hsu, D., Kale, S., Langford, J., Li, L., & Schapire, R. (2014). Taming the monster: A fast and simple algorithm for contextual bandits. In *Proceedings of the 31th international conference on machine learning, icml* (Vol. 32, *2*, pp. 1638–1646). JMLR Workshop and Conference Proceedings. PMLR.

Agrawal, R. (1995). The continuum-armed bandit problem. *SIAM journal on control and optimization*, *33*(6), 1926–1951.

Agrawal, S., & Goyal, N. (2012). Analysis of thompson sampling for the multi-armed bandit problem. In S. Mannor, N. Srebro, & R. C. Williamson (Eds.), *Proceedings of the 25th annual conference on learning theory, COLT* (Vol. 23, pp. 39.1–39.26). Proceedings of Machine Learning Research. PMLR.

Agrawal, S., & Goyal, N. (2013a). Further optimal regret bounds for thompson sampling. In C. M. Carvalho & P. Ravikumar (Eds.), *Proceedings of the sixteenth international conference on artificial intelligence and statistics, AISTATS* (Vol. 31, pp. 99–107). Proceedings of Machine Learning Research. Scottsdale, Arizona, USA: PMLR.

Agrawal, S., & Goyal, N. (2013b). Thompson sampling for contextual bandits with linear payoffs. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (Vol. 28, *3*, pp. 127–135). Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR.

Amazon. (2012). *Amazon Mechanical Turk*. Retrieved from https://www.mturk.com/

Ansari, A., & Mela, C. F. (2003). E-customization. *Journal of Marketing Research*, *40*(2), 131–145. doi:10.1509/jmkr.40.2.131.19224

Audibert, J., Bubeck, S., & Munos, R. (2010). Best arm identification in multi-armed bandits. In A. T. Kalai & M. Mohri (Eds.), *Proceedings of the 23rd conference on learning theory, COLT* (pp. 41–53). Omnipress.

Audibert, J.-Y., Munos, R., & Szepesvári, C. (2009). Exploration–exploitation trade-off using variance estimates in multi-armed bandits. *Theoretical Computer Science*, *410*(19), 1876–1902. doi:10.1016/j.tcs.2009.01.016

Auer, P. (2002). Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, *3*(Nov), 397–422.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multi-armed bandit problem. *Machine Learning*, *47*(2-3), 235–256.

Auer, P., & Ortner, R. (2010). UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, *61*(1–2), 55–65. doi:10.1007/s10998-010-3055-6

Austin, P. C. (2011). An introduction to propensity score methods for reducing the effects of confounding in observational studies. *Multivariate Behavioral Research*, *46*(3), 399–424. doi:10.1080/00273171.2011.568786

Bai, X., Guan, J., & Wang, H. (2019). A model-based reinforcement learning with adversarial training for online recommendation. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32, NeurIPS* (pp. 10734–10745).

Bakshy, E., Dworkin, L., Karrer, B., Kashin, K., Letham, B., Murthy, A., & Singh, S. (2018). AE: A domain-agnostic platform for adaptive experimentation. In *Neurips: Systems for ml workshop*.

Bakshy, E., Eckles, D., & Bernstein, M. S. (2014). Designing and deploying online field experiments. In C. Chung, A. Z. Broder, K. Shim, & T. Suel (Eds.), *Proceedings of the 23rd international world wide web conference, WWW* (pp. 283–292). doi:10.1145/2566486.2567967

Bastani, H., & Bayati, M. (2020). Online decision-making with high-dimensional covariates. *Operations Research*, *68*(1), 276–294. doi:10.1287/opre.2019.1902

Baumeister, H., Reichler, L., Munzinger, M., & Lin, J. (2014). The impact of guidance on internet-based mental health interventions—a systematic review. *Internet Interventions*, *1*(4), 205–215.

Berry, D. A. (2006). Bayesian clinical trials. *Nat Rev Drug Discov*, *5*(1), 27–36. doi:10.1038/nrd1927

Berry, D. A. (2012). Adaptive clinical trials in oncology. *Nat Rev Clin Oncol*, *9*(4), 199–207. doi:10.1038/nrclinonc.2011.165

Berry, D. A., & Fristedt, B. (1985). *Bandit problems: Sequential allocation of experiments*. doi:10.1007/978-94-015-3711-7

Beygelzimer, A., Hsu, D. J., Langford, J., & Zhang, T. (2010). Agnostic active learning without constraints. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, & A. Culotta (Eds.), *Advances in Neural Information Processing Systems 23, NeurIPS* (pp. 199–207).

Beygelzimer, A., Langford, J., Li, L., Reyzin, L., & Schapire, R. E. (2011). Contextual bandit algorithms with supervised learning guarantees. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics, aistats* (pp. 19–26).

Bietti, A., Agarwal, A., & Langford, J. (2018). A contextual bandit bake-off. eprint: arXiv:18-02.04064

Bouneffouf, D., & Rish, I. (2019). A survey on practical applications of multi-armed and contextual bandits. eprint: arXiv:1904.10040

Bouneffouf, D., Rish, I., & Cecchi, G. A. (2017). Bandit models of human behavior: Reward processing in mental disorders. In T. Everitt, B. Goertzel, & A. Potapov (Eds.), *Proceedings of the 10th International Conference on Artificial General Intelligence, AGI* (Vol. 10414, pp. 237–248). Lecture Notes in Computer Science. doi:10.1007/978-3-319-63703-7_22

Box, G. E., & Tiao, G. C. (1992). *Bayesian inference in statistical analysis*. John Wiley & Sons.

Bubeck, S., & Cesa-Bianchi, N. (2012). Regret analysis of stochastic and non-stochastic multi-armed bandit problems. *Foundations and Trends in Machine Learning*, *5*(1), 1–122. doi:10.1561/2200000024

Bubeck, S., Munos, R., & Stoltz, G. (2011). Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, *412*(19), 1832–1852. doi:10.1016/j.tcs.2010.12.059

Buytaert, D. et al. (2001). *Drupal*. Retrieved from https://drupal.org

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., ... Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of Statistical Software*, *76*(1), 1–32. doi:10.18637/jss.v076.i01

Chapelle, O., & Li, L. (2011). An empirical evaluation of thompson sampling. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. C. N. Pereira, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 24, NeurIPS* (pp. 2249–2257).

Chebli, J.-L., Blaszczynski, A., & Gainsbury, S. M. (2016). Internet-based interventions for addictive behaviours: A systematic review. *Journal of Gambling Studies*, *32*(4), 1279–1304. doi:10.1007/s10899-016-9599-5

Cheng, H., & Cantú-Paz, E. (2010). Personalized click prediction in sponsored search. In *Proceedings of the third acm international conference on web search* *and data mining, wsdm* (pp. 351–360). WSDM ’10. doi:10.1145/1718487.1718531

Chernoff, H. (1972). *Sequential analysis and optimal design*. SIAM.

Clement, B., Roy, D., Oudeyer, P., & Lopes, M. (2015). Multi-armed bandits for intelligent tutoring systems. In O. C. Santos, J. Boticario, C. Romero, M. Pechenizkiy, A. Merceron, P. Mitros, . . . M. C. Desmarais (Eds.), *Proceedings of the 8th International Conference on Educational Data Mining, EDM* (p. 21). IEDMS.

Coffey, C. S., & Kairalla, J. A. (2008). Adaptive clinical trials. *Drugs in R & D*, *9*(4), 229–242.

Collet, J., Sassolas, T., Lhuillier, Y., Sirdey, R., & Carlier, J. (2016). Leveraging distributed graphlab for program trace analysis. In *2016 international conference on high performance computing & simulation, HPCS*. doi:10.1109/hpcsim.2016.7568341

Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., & Xing, E. P. (2016). GeePS: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the eleventh european conference on computer systems, eurosys*. doi:10.1145/2901318.2901323

De, B. (2017). API testing strategy. In *API management* (pp. 153–164). doi:10.1007/978-1-4842-1305-6_9

Djolonga, J., Krause, A., & Cevher, V. (2013). High-dimensional gaussian process bandits. In C. J. C. Burges, L. Bottou, Z. Ghahramani, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems 26, NeurIPS* (pp. 1-025–1033).

Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. In *Sequential monte carlo methods in practice* (pp. 3–14). Springer-Verlag.

Dropp, K. A. (2014). Implementing a conjoint analysis design in qualtrics. *(Working Paper). Dartmouth College*.

Dudík, M., Erhan, D., Langford, J., & Li, L. (2012). Sample-efficient nonstationary policy evaluation for contextual bandits. In N. de Freitas & K. P. Murphy (Eds.), *Proceedings of the 28th Conference on Uncertainty in Artificial, UAI* (pp. 247–254). AUAI Press.

Dudík, M., Erhan, D., Langford, J., Li, L., et al. (2014). Doubly robust policy evaluation and optimization. *Statistical Science*, *29*(4), 485–511. doi:10.1214/14-STS500

Dudík, M., Hsu, D. J., Kale, S., Karampatziakis, N., Langford, J., Reyzin, L., & Zhang, T. (2011). Efficient optimal learning for contextual bandits. In F. G.

Cozman & A. Pfeffer (Eds.), *Proceedings of the 27th Conference on Uncertainty, UAI* (pp. 169–178). AUAI Press.

Dudík, M., Langford, J., & Li, L. (2011). Doubly robust policy evaluation and learning. In L. Getoor & T. Scheffer (Eds.), *Proceedings of the 28th International Conference on Machine Learning, ICML* (pp. 1097–1104).

Durand, A., Achilleos, C., Iacovides, D., Strati, K., Mitsis, G. D., & Pineau, J. (2018). Contextual bandits for adapting treatment in a mouse model of de novo carcinogenesis. In F. Doshi-Velez, J. Fackler, K. Jung, D. C. Kale, R. Ranganath, B. C. Wallace, & J. Wiens (Eds.), *Proceedings of the machine learning for healthcare conference, mlhc* (Vol. 85, pp. 67–82). Proceedings of Machine Learning Research. PMLR.

Eckles, D., & Kaptein, M. (2014). Thompson sampling with the online bootstrap. eprint: arXiv:1410.4009

Eckles, D., & Kaptein, M. (2019). Bootstrap thompson sampling and sequential decision problems in the behavioral sciences. *SAGE Open*, *9*(2). doi:10.1177/2158244019851675

Efron, B., & Morris, C. (1975). Data analysis using stein's estimator and its generalizations. *Journal of the American Statistical Association*, *70*(350), 311–319. doi:10.1080/01621459.1975.10479864

Fisher, R. A. (1990). *Statistical methods, experimental design, and scientific inference*. Oxford Univ. Press.

Galbraith, B. (2016). *Python library for multi-armed bandits*. Retrieved from https://github.com/bgalbraith/bandits

Garivier, A., & Cappé, O. (2011). The KL-UCB algorithm for bounded stochastic bandits and beyond. In S. M. Kakade & U. von Luxburg (Eds.), *Proceedings of the 24th Annual Conference on Learning Theory, COLT* (Vol. 19, pp. 359–376). JMLR Proceedings. PMLR.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., & Rubin, D. B. (2013). *Bayesian Data Analysis, Third Edition*. CRC Press.

Gelman, A., & Hill, J. (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Gittins, J. C. (1989). *Multi-armed bandit allocation indices*. Wiley.

Google. (2018). *Optimize, Google Analytics Solutions*. Retrieved from https://www.google.com/analytics/optimize/features/

Goos, P., & Jones, B. (2011). *Optimal design of experiments: A case study approach*. John Wiley & Sons.

Hanneke, S. (2014). Theory of disagreement-based active learning. *Foundations and Trends in Machine Learning*, *7*(2–3), 131–309. doi:10.1561/2200000037

Hardwick, J., Oehmke, R., & Stout, Q. F. (1999). A program for sequential allocation of three bernoulli populations. *Computational Statistics & Data Analysis*, *31*(4), 397–416. doi:10.1016/S0167-9473(99)00039-0

Hauser, J. R., Urban, G. L., Liberali, G., & Braun, M. (2009). Website Morphing. *Marketing Science*, *28*(2), 202–223. doi:10.1287/mksc.1080.0459

Hido, S., Tokui, S., & Oda, S. (2013). Jubatus: An open source platform for distributed online machine learning. In *Neurips 2013 workshop on big learning*.

Hirano, K., & Imbens, G. W. (2004). The propensity score with continuous treatments. In A. Gelman & X.-L. Meng (Eds.), *Applied bayesian modeling and causal inference from incomplete-data perspectives: An essential journey with donald rubin's statistical family* (pp. 73–84). doi:10.1002/0470090456.ch7

Horvitz, D. G., & Thompson, D. J. (1952). A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, *47*(260), 663–685. doi:10.1080/01621459.1952.10483446

Hsu, J. (1996). *Multiple comparisons: Theory and methods*. CRC Press.

Huber, J., Payne, J. W., & Puto, C. (1982). Adding asymmetrically dominated alternatives: Violations of regularity and the similarity hypothesis. *Journal of Consumer Research*, *9*(1), 90–98. doi:10.1086/208899

Ierusalimschy, R. (2016). *Programming in Lua* (4th). Lua.org.

Imbens, G. W., & Rubin, D. B. (2015). *Causal inference for statistics, social, and biomedical sciences*. doi:10.1017/cbo9781139025751

iPerf Authors. (2016). *Iperf – the ultimate speed test tool for tcp, udp and sctp*. Retrieved from https://iperf.fr/

Ippel, L., Kaptein, M., & Vermunt, J. (2016a). Dealing with data streams. *Methodology*, *12*(4), 124–138. doi:10.1027/1614-2241/a000116

Ippel, L., Kaptein, M., & Vermunt, J. (2019). Online estimation of individual-level effects using streaming shrinkage factors. *Computational Statistics & Data Analysis*, *137*, 16–32. doi:10.1016/j.csda.2019.01.010

Ippel, L., Kaptein, M., & Vermunt, J. K. (2016b). Estimating random-intercept models on data streams. *Computational Statistics & Data Analysis*, *104*, 169–182. doi:10.1016/j.csda.2016.06.008

James, W., & Stein, C. (1961). Estimation with quadratic loss. In J. Neyman (Ed.), *Proceedings of the fourth berkeley symposium on mathematical statistics and probability* (Vol. 1, pp. 361–379). doi:10.1177/0956797611430953

Jamieson, K. G., & Jain, L. (2018). A bandit approach to sequential experimental design with false discovery control. In S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31, NeurIPS* (pp. 3664–3674).

Jamieson, K. G., & Nowak, R. D. (2014). Best-arm identification algorithms for multi-armed bandits in the fixed confidence setting. In *Proceedings of the 48th annual conference on information sciences and systems, CISS* (pp. 1–6). doi:10.1109/CISS.2014.6814096

Javanmard, A., & Nazerzadeh, H. (2019). Dynamic pricing in high-dimensions. *The Journal of Machine Learning Research*, *20*(1), 315–363.

Jiang, B., Shi, X., Shang, H., Geng, Z., & Glass, A. (2016). A framework for network ab testing. eprint: {arXiv}:1610.07670

Johari, R., Pekelis, L., & Walsh, D. J. (2015). Always valid inference: Bringing sequential analysis to a/b testing. eprint: arXiv:1512.04922

Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285. doi:10.1613/jair.301

Kaibel, C., & Biemann, T. (2019). Rethinking the gold standard with multi-armed bandits: Machine learning allocation algorithms for experiments. *Organizational Research Methods*. doi:10.1177/1094428119854153

Kallus, N., & Zhou, A. (2018). Policy evaluation and optimization with continuous treatments. eprint: arXiv:1802.06037

Kaptein, M. (2014). The use of thompson sampling to increase estimation precision. *Behavior Research Methods*, *47*(2), 409–423. doi:10.3758/s13428-014-0480-0

Kaptein, M., De Ruyter, B., Markopoulos, P., & Aarts, E. (2012). Adaptive persuasive systems: A study of tailored persuasive text messages to reduce snacking. *ACM Transactions on Interactive Intelligent Systems*, *2*(2), 1–25. doi:10.1145/2209310.2209313

Kaptein, M., & Ianuzzi, D. (2016). Lock in feedback in sequential experiments. eprint: arXiv:1502.00598

Kaptein, M., Markopoulos, P., De Ruyter, B., & Aarts, E. (2015). Personalizing persuasive technologies: Explicit and implicit personalization using persuasion profiles. *International Journal of Human-Computer Studies*, *77*, 38–51. doi:10.1016/j.ijhcs.2015.01.004

Kaptein, M., McFarland, R., & Parvinen, P. (2018). Automated adaptive selling. *European Journal of Marketing*, *52*(5/6), 1037–1059. doi:10.1108/EJM-08-2016-0485

Kaptein, M., Van Emden, R., & Iannuzzi, D. (2017). Uncovering noisy social signals: Using optimization methods from experimental physics to study social phenomena. *PLOS ONE*, *12*(3). doi:10.1371/journal.pone.0174182

Kaptein, M., van Emden, R., & Iannuzzi, D. (2016a). Investigation of the concept of beauty via a lock-in feedback experiment. eprint: arXiv:1607.08108

Kaptein, M., van Emden, R., & Iannuzzi, D. (2016b). Tracking the decoy: Maximizing the decoy effect through sequential experimentation. *Palgrave Communications*, *2*(1). doi:10.1057/palcomms.2016.82

Kaufmann, E., Cappé, O., & Garivier, A. (2012). Pymabandits: Matlab and python packages for multi-armed bandits. Retrieved from http://mloss.org/software/view/415/

Kaufmann, E., Cappé, O., & Garivier, A. (2016). On the complexity of best-arm identification in multi-armed bandit models. *The Journal of Machine Learning Research*, *17*(1), 1–42.

Kaufmann, E., Korda, N., & Munos, R. (2012). Thompson sampling: An asymptotically optimal finite-time analysis. In N. H. Bshouty, G. Stoltz, N. Vayatis, & T. Zeugmann (Eds.), *Algorithmic learning theory* (pp. 199–213). Lecture Notes in Computer Science. doi:10.1007/978-3-642-34106-9_18

Kleinberg, R. D. (2004). Nearly tight bounds for the continuum-armed bandit problem. In *Advances in neural information processing systems 17, neurips* (pp. 697–704).

Klenov, K. (2015). *Python's web framework benchmarks*. Retrieved from http://klen.github.io/py-frameworks-bench/

Komiyama, J., Honda, J., & Nakagawa, H. (2015). Optimal regret analysis of thompson sampling in stochastic multi-armed bandit problem with multiple plays. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd International Conference on Machine Learning, ICML* (Vol. 37, pp. 1152–1161). Proceedings of Machine Learning Research.

Krause, A., & Ong, C. S. (2011). Contextual gaussian process bandit optimization. In *Advances in Neural Information Processing Systems 24, NeurIPS* (pp. 2447–2455).

Kruijswijk, J., Parvinen, P., & Kaptein, M. (2019). Exploring offline policy evaluation for the continuous-armed bandit problem. eprint: arXiv:1908.07808

Kruijswijk, J., van Emden, R., Parvinen, P., & Kaptein, M. (2020). StreamingBandit; experimenting with bandit policies. *Journal of Statistical Software*.

Kuleshov, V., & Precup, D. (2014). Algorithms for multi-armed bandit problems. eprint: arXiv:1402.6028

Lachin, J. M., Matts, J. P., & Wei, L. J. (1988). Randomization in clinical trials: Conclusions and recommendations. *Controlled Clinical Trials*, *9*(4), 365–374. doi:10.1016/0197-2456(88)90049-9

Lai, T., & 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. (2005). Tutorial on practical prediction theory for classification. *Journal of Machine Learning Research*, *6*, 273–306.

Langford, J., Li, L., & Strehl, A. (2011). *Vowpal wabbit*. Retrieved from https://www.vowpalwabbit.org

Langford, J., & Zhang, T. (2008). The epoch-greedy algorithm for multi-armed bandits with side information. In *Advances in neural information processing systems 20, neurips* (pp. 817–824).

Lattimore, T., & Szepesvári, C. (2020). *Bandit algorithms*. Cambridge University Press.

Li, L., Chu, W., Langford, J., & Schapire, R. E. (2010). A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on world wide web, www* (pp. 661–670). 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* (pp. 297–306). doi:10.1145/1935826.1935878

Libin, P. J., Verstraeten, T., Roijers, D. M., Grujic, J., Theys, K., Lemey, P., & Nowé, A. (2018). Bayesian best-arm identification for selecting influenza mitigation strategies. In *Joint european conference on machine learning and knowledge discovery in databases* (pp. 456–471). Springer.

Macready, W. G., & Wolpert, D. H. (1998). Bandit problems and the exploration/-exploitation tradeoff. *IEEE Transactions on Evolutionary Computation*, *2*(1), 2–22. doi:10.1109/4235.728210

Mandelbaum, A. (1987). Continuous multi-armed bandits and multiparameter processes. *The Annals of Probability*, *15*(4), 1527–1556. doi:10.1214/aop/1176991992

Marrero, M., & Hauff, C. (2018). A/b testing with APONE. In *The 41st international acm sigir conference on research & development in information retrieval* (pp. 1269–1272). doi:10.1145/3209978.3210164

Mary, J., Preux, P., & Nicol, O. (2014). Improving offline evaluation of contextual bandit algorithms via bootstrapping techniques. In *Proceedings of the 31th international conference on machine learning, icml* (Vol. 32, pp. 172–180). JMLR Workshop and Conference Proceedings. JMLR.org.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., ... Talwalkar, A. (2016). MLlib: Machine learning in apache spark. *Journal of Machine Learning Research*, *17*(34), 1–7.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

Michalak, S., DuBois, A., DuBois, D., Wiel, S. V., & Hogden, J. (2012). Developing systems for real-time streaming analysis. *Journal of Computational and Graphical Statistics, 21*(3), 561–580. doi:10.1080/10618600.2012.657144

Mixpanel. (2017). *Mixpanel: Product analytics for product people*. Retrieved from https://mixpanel.com

Murnaghan, D. A., Sihvonen, M., Leatherdale, S. T., & Kekki, P. (2007). The relationship between school-based smoking policies and prevention programs on smoking behavior among grade 12 students in prince edward island: A multilevel analysis. *Preventive Medicine, 44*(4), 317–322. doi:10.1016/j.ypmed.2007.01.003

Nugent, T. (2015). *Epsilon-greedy, softmax and linucb contextual bandit implementations*. Retrieved from https://github.com/timnugent/bandit-algorithms

Optimizely. (2017). *Optimizely*. Retrieved from https://www.optimizely.com

Osband, I., & Roy, B. V. (2015). Bootstrapped thompson sampling and deep exploration. eprint: arXiv:1507.00300

Owen, A. B., & Eckles, D. (2012). Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics*, *6*(3), 895–927. doi:10.1214/12-aoas547

Pandey, S., Agarwal, D., Chakrabarti, D., & Josifovski, V. (2007). Bandits for taxonomies: A model-based approach. In *Proceedings of the seventh SIAM international conference on data mining, april 26-28, 2007, minneapolis, minnesota, USA* (pp. 216–227). doi:10.1137/1.9781611972771.20

Pandey, S., Chakrabarti, D., & Agarwal, D. (2007). Multi-armed bandit problems with dependent arms. In *Proceedings of the 24th international conference on machine learning*. doi:10.1145/1273496.1273587

Parvin, P., Chessa, S., Kaptein, M., & Paternò, F. (2019). Personalized real-time anomaly detection and health feedback for older adults. *Journal of Ambient Intelligence and Smart Environments, 11*(5), 453–469. doi:10.3233/ais-190536

Pearl, J. (2009). *Causality*. doi:10.1017/cbo9780511803161

Press, W. H. (2009). Bandit solutions provide unified ethical models for randomized clinical trials and comparative effectiveness research. In *Proceedings of the national academy of sciences* (Vol. 106, pp. 22387–22392). doi:10.1073/pnas.0912378106

Qualtrics. (2005). *Qualtrics*. Retrieved from https://www.qualtrics.com

R Core Team. (2019). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria. Retrieved from https://www.R-project.org/

Reagen, B., Whatmough, P., Adolf, R., Rama, S., Lee, H., Lee, S. K., . . . Brooks, D. (2016). Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 2016 ACM/IEEE 43rd annual international symposium on computer architecture, isca*. doi:10.1109/isca.2016.32

Robbins, H. (1952). Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, *58*(5), 527–535.

Russo, D. (2016). Simple bayesian algorithms for best arm identification. In V. Feldman, A. Rakhlin, & O. Shamir (Eds.), *Proceedings of the 29th conference on learning theory, colt* (Vol. 49, pp. 1417–1418). JMLR Workshop and Conference Proceedings. JMLR.org.

Ryan, E. G., Drovandi, C. C., McGree, J. M., & Pettitt, A. N. (2016). A review of modern computational algorithms for bayesian optimal design. *International Statistical Review*, *84*(1), 128–154. doi:10.1111/insr.12107

Schwartz, E. M., Bradlow, E. T., & Fader, P. S. (2017). Customer acquisition via display advertising using multi-armed bandit experiments. *Marketing Science*, *36*(4), 500–522. doi:10.1287/mksc.2016.1023

Scofield, J. H. (1994). Frequency-domain description of a lock-in amplifier. *American Journal of Physics*, *62*(2), 129. doi:10.1119/1.17629

Scott, S. L. (2010). A modern Bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, *26*(6), 639–658. doi:10.1002/asmb.873

Seide, F., & Agarwal, A. (2016). Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM international conference on knowledge discovery and data mining, sigkdd*. doi:10.1145/2939672.2945397

Sola, D. (2015). *Bandit algorithms and test framework in java*. Retrieved from https://github.com/danisola/bandit

Stein, C. (1956). Inadmissibility of the Usual Estimator of the Mean of a Multivariate Normal Distribution. In *Proceedings of the third berkeley symposium on mathematical statistics and probability.* (pp. 197–206). Berkeley: Univ. Cal. Press.

Striatum Contributors. (2016). *Contextual bandit in python*. Retrieved from https://github.com/ntucllab/striatum

SurveyMonkey. (1999). *Surveymonkey surveys*. Retrieved from https://www.surveymonkey.com

Sutton, R. S., & Barto, A. G. (2011). *Reinforcement learning: An introduction*. MIT Press.

Szepesvári, C. (2010). Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, *4*(1), 1–103. doi:10.2200/s00268ed1v01y201005aim009

Tene, G. (2015). *Wrk2: A constant throughput, correct latency recording variant of wrk*. Retrieved from https://github.com/giltene/wrk2

Tewari, A., & Murphy, S. A. (2017). From ads to interventions: Contextual bandits in mobile health. In J. Rehg, S. A. Murphy, & S. Kumar (Eds.), *Mobile health* (pp. 495–517). doi:10.1007/978-3-319-51394-2_25

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, *25*(3/4), 285–294. doi:10.2307/2332286

Tornado Authors. (2016). *Tornado software version 4.4.2*. Retrieved from http://www.tornadoweb.org

van den Berg, M., Schoones, J., & Vlieland, T. V. (2007). Internet-based physical activity interventions: A systematic review of the literature. *Journal of Medical Internet research*, *9*(3), e26. doi:10.2196/jmir.9.3.e26

van Emden, R., & Kaptein, M. (2018). Contextual: Evaluating contextual multi-armed bandit problems in R. eprint: arXiv:1811.01926

Vaswani, S., Kveton, B., Wen, Z., Ghavamzadeh, M., Lakshmanan, L. V., & Schmidt, M. (2017). Model-independent online learning for influence maximization. In *Proceedings of the 34th international conference on machine learning, icml* (pp. 3530–3539). JMLR. org.

Villar, S. S., Bowden, J., & Wason, J. (2015). Multi-armed bandit models for the optimal design of clinical trials: Benefits and challenges. *Statistical Science*, *30*(2), 199–215. doi:10.1214/14-sts504

Wang, C.-C., Kulkarni, S. R., & Poor, H. V. (2005). Bandit problems with side observations. *IEEE Transactions on Automatic Control*, *50*(3), 338–355. doi:10.1109/TAC.2005.844079

Wang, Y., Agarwal, A., & Dudík, M. (2017). Optimal and adaptive off-policy evaluation in contextual bandits. In D. Precup & Y. W. Teh (Eds.), *Proceedings of the 34th International Conference on Machine Learning, ICML* (Vol. 70, pp. 3589–3597). Proceedings of Machine Learning Research. PMLR.

Weber, S. (2019). A step-by-step procedure to implement discrete choice experiments in qualtrics. *Social Science Computer Review*. doi:10/ghdcxn

Welford, B. P. (1962). Note on a method for calculating corrected sums of squares and products. *Technometrics*, *4*(3), 419–420. doi:10.2307/1266577

Whitehead, J., & Brunier, H. (1995). Bayesian decision procedures for dose determining experiments. *Statistics in Medicine*, *14*(9), 885–893. doi:10.1002/sim.4780140904

Whittle, P. (1980). Multi-armed bandits and the gittins index. *Journal of the Royal Statistical Society B*, *42*(2), 143–149. doi:10/fm77

Williams, J. J., Rafferty, A. N., Maldonado, S., Ang, A., Tingley, D., & Kim, J. (2017). Mooclets: A framework for dynamic experimentation and personalization. In *Proceedings of the fourth acm conference on learning@ scale, l@s* (pp. 287–290). doi:10.1145/3051457.3054006

Williamson, S. F., Jacko, P., Villar, S. S., & Jaki, T. (2017). A bayesian adaptive design for clinical trials in rare diseases. *Computational Statistics & Data Analysis*, *113*, 136–153. doi:10.1016/j.csda.2016.09.006

Wolfson, R. (1991). The lock-in amplifier: A student experiment. *Am. J. Phys*, *59*(6), 569–572. doi:10.1119/1.16824

Yang, F., Ramdas, A., Jamieson, K. G., & Wainwright, M. J. (2017). A framework for multi-a(rmed)/b(andit) testing with online FDR control. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30, NeurIPS* (pp. 5957–5966).

Yelp. (2014). *Moe: Metric optimization engine*. Retrieved from https://github.com/Yelp/MOE

Summary

In experiments that consider the use of subjects, a crucial part is deciding which treatment to allocate to which subject – in other words, constructing the treatment allocation procedure. In a classical experiment, this treatment allocation procedure often simply constitutes randomly assigning subjects to a number of different treatments. Subsequently, when all outcomes have been observed, the resulting data is used to conduct an analysis that is specified a priori. Practically, however, the subjects often arrive at an experiment one-by-one. This allows the data generating process to be viewed differently: instead of considering the subjects in a batch, intermediate data from previous interactions with other subjects can be used to influence the decisions of the treatment allocation in future interactions. A heavily researched formalization that helps developing strategies for sequentially allocating subjects is the multi-armed bandit problem. In this thesis, methods to expedite the use of sequential allocation procedures are developed. This is done by building upon the extensive literature of the multi-armed bandit problem. The thesis also introduces and shows many (empirical) examples of the usefulness and applicability of sequential allocation procedures in practice.

Although the multi-armed bandit literature provides a formalization for sequential allocation procedures, it is still notoriously hard to develop and evaluate policies – as treatment allocation procedures are called in the MAB literature – that tackle applied instances of multi-armed bandit problems and to apply those policies in studies. The thesis addresses some of the problems involved. Chapter 2 introduces a framework and a software application, called **StreamingBandit**, to address some of the issues of developing and evaluating MAB policies to apply them in experiments. The framework identifies how to formally summarize sequential allocation procedures into two steps, which helps implementing any allocation procedure in the software proposed in this chapter. The user-friendly software for developing, evaluating and deploying sequential allocation procedures on the web is introduced with detailed explanations. Furthermore, various examples showing how to create and implement sequential allocation procedures are presented.

Chapter 3 introduces various methods for dealing with nested data structures

in sequential allocation procedures and applies these methods to several popular policies. One example – the one that is considered in this chapter and is very common in the social sciences – of a nested data structure occurs when subjects are returning for multiple interactions. The observations are then nested within subjects over time and are thus dependent observations. The traditional statistical literature already provides us with methods to deal with such dependencies – which are all too often ignored in the multi-armed bandit literature. Two methods that are considered here are a shrinkage factor approach and a Bayesian hierarchical modeling approach. Through simulations and an empirical study it is shown that taking such hierarchical dependencies into account greatly improves the performance of sequential allocation procedures when these dependencies are present.

To evaluate and validate sequential allocation procedures in the field without resorting to multiple field evaluations or to simulation based methods that lack external validity, so-called offline evaluation can be performed. Offline evaluation methods use data collected in the field to evaluate sequential allocation procedures. Currently, these methods are developed mostly for experiments that consider discrete treatments; the continuous case is hardly covered. Chapter 4 introduces a new method to evaluate sequential allocation procedures for continuous treatments. The proposed delta method is compared to a recently introduced method that considers only static procedures, in which model (or policy) parameters are not updated based on the previous interactions. The chapter details several simulation and empirical studies to evaluate the proposed method and shows that it achieves relative consistent ranking of sequential allocation procedures in offline evaluations.

Chapter 5 ensures that the results presented in this thesis can be used by researchers in the social sciences: it shows step-by-step how to integrate **StreamingBandit** into a front-end web application to implement sequential treatment allocation procedures. Several currently available web applications allow researchers to do experiments through the web, but they are often limited to very simple treatment allocation procedures such as simple uniform random treatment allocation. However, by integrating **StreamingBandit** within these web applications, it becomes possible to use much more advanced allocation procedures, and truly benefit from the results presented in this thesis. This is illustrated using an application with **Qualtrics**, an online platform for conducting experiments and surveys. We show that once **StreamingBandit** is integrated within a **Qualtrics** application, it becomes easy to modify and thus experiment with different allocation procedures. Furthermore, the chapter gives an illustration of a recently completed experiment using **StreamingBandit** with sequential allocation procedures.

Concluding, in Chapter 6 the contributions and limitations of the research in this thesis are discussed. The chapter also discusses considerations that can be taken when using sequential allocation procedures for hypothesis testing, best-arm (or treatment) identification and for the use of the optimal design of experiments. Finally, future research directions are also provided, such as improving the outreach and user-friendliness of **StreamingBandit** and putting the methods developed in Chapter 3 and 4 under more theoretical scrutiny.

Acknowledgements

Dit proces had ik niet zomaar tot een goed einde kunnen brengen zonder de hulp van vele anderen die ik daarvoor wil bedanken.

Voor ik begin wil ik graag dr. Louis Vuurpijl bedanken. Toen hij mij tijdens de eerste dagen van mijn master, in september 2014, aan Maurits introduceerde als mogelijk nieuwe student-assistent, heeft hij het begin van onze samenwerking in gang gezet. Helaas heeft hij daarna niet meer mogen meemaken wat die introductie heeft opgeleverd. Ik had hem graag dit boekje willen overhandigen.

Maurits, de afgelopen zes jaar hebben wij steevast samengewerkt. Eerst was ik je student-assistent, toen nog in Nijmegen. Daarna had ik de eer om mee te verhuizen naar Tilburg (dat was voor mij gelukkig een stuk minder ver weg), waar je me de kans bood om onder jouw begeleiding (en die van Jeroen) te promoveren. Ik heb je manier van begeleiden altijd erg gewaardeerd. Je liet de teugels vieren waar het kon, maar liet het altijd weten als je vond dat het anders kon en moest. Je was altijd bereikbaar, serieus, kritisch en open, maar daarbij vergat je nooit dat er bovenal plezier gemaakt moest worden. Je hebt me veel geleerd en liet me kennis maken met veel nieuwe dingen: sterke papers schrijven, projecten uitvoeren bij grote internationale bedrijven, ondernemend denken, actief een netwerk opbouwen en je verhaal overbrengen. Op al deze gebieden ben je echt een voorbeeld. Bedankt voor alles.

Jeroen, ik wil je bedanken voor de sturing en alle feedback van de afgelopen vier jaar. Je stond altijd klaar met advies op momenten dat het nodig was. Bedankt voor het vertrouwen en de vrijheid die je daarbij hebt gegeven en voor het bieden van deze kans.

Ook wil ik graag prof. Loes Keijsers, prof. Elena Marchiori, prof. Eric Postma en prof. Arjen de Vries bedanken voor het lezen en beoordelen van mijn proefschrift en voor het e-reizen naar mijn verdediging.

Petri, thanks for hosting me more than once in Finland, for providing me with the opportunities of working with different companies and teaching me that there are many ways to skin the academic cat. I had a great time every single visit, it really inspired me to think out of the box.

Niek, Eva, Inga, bedankt voor het op peil houden van mijn geestelijke gezondheid tijdens de vele wandelingen. Jullie waren een fijn klankbord en ik ben blij met de band die we hebben opgebouwd. Mijn collega's van MTO, bedankt voor de gezelligheid tijdens de vele (potluck) lunches, borrels en andere uitjes. In het bijzonder nog mijn kamergenoten Elise, Erwin, Felix, Hilde en Zhengguo, bedankt voor de goede gesprekken, de small talk en de altijd gevulde snoeppot. Door jullie allemaal heb ik in kunnen zien hoe belangrijk het is om samen te kunnen werken met fijne collega's.

Mijn labgroep bij JADS: Bas, Frederique, Florian, Hongyi, Lingjie, Robin, Thomas, Xynthia, Ylva en Zoltán. Bedankt voor alle feedback op de vele versies van mijn papers en de leuke tijd in Den Bosch. In het bijzonder Florian en Robin voor alle hulp tijdens de afgelopen vier jaar. Ik vond het een eer om samen met jullie aan verschillende projecten te werken en jullie hebben me veel interessante nieuwe inzichten gegeven.

De groep bij CZ, Fleur en consorten. Ondanks dat we, helaas, niet tot een vruchtbaar onderzoek hebben kunnen komen, wil ik jullie bedanken voor de tijd die ik bij jullie door mocht brengen. Er was naast het serieuze werken altijd tijd voor gezelligheid. Ik heb bij jullie een beetje mogen proeven hoe het is om in een team te werken en daar heb ik veel van geleerd.

Ik heb veel fijne en lieve vrienden die steeds weer voor mij klaarstaan. Lars wil ik als vormgever van dit proefschrift hiervoor in het bijzonder bedanken. Je hebt echt smoel gegeven aan de inhoud, daar ben ik heel trots op.

Lieve familie achterop, Anita, Ad, Anne, Bram, Peter, Saskia, Ronald, Renske, Guusje en René, bedankt voor de ontspanning, gezelligheid en de warmte die jullie mij altijd geven. Jullie zijn altijd geïnteresseerd en ik kan altijd mijn verhaal bij jullie kwijt, dat is een fijne basis om op terug te kunnen vallen.

Lieve papa en mama, bedankt voor alle onvoorwaardelijke steun van de afgelopen 28 jaar. Jullie staan altijd voor me klaar en hebben me altijd vrij laten ontdekken, waarvoor ik jullie altijd dankbaar zal zijn. En bedankt dat jullie samen met Émile, Léon, Marie-Noëlle en Nina altijd het geduld en het vertrouwen hebben gehad dat ik mijn 'mentale beperking', die juf Lilian mij toedichtte, ooit te boven zou komen.

Als allerlaatste wil ik de belangrijkste persoon in mijn leven bedanken. Lieve Nadia, zonder jouw onuitputtelijke liefde en steun van de afgelopen jaren was ik nooit zo ver gekomen. Ik ben heel blij jou mijn vrouw te mogen noemen. Met jou kan ik echt overal over praten en je helpt me door alle moeilijke momenten heen te komen, niet alleen die van mijn promoveren. De tijd heeft geleerd dat we samen alles te boven kunnen komen. Mijn proefschrift is daardoor minstens voor de helft dat van jou. Ik hou van je.