`STREAMINGBANDIT`: DEVELOPING ADAPTIVE PERSUASIVE SYSTEMS.

MAURITS KAPTEIN AND JULES KRUIJSWIJK

Abstract. This paper introduces `StreamingBandit`, a (back-end) solution for developing adaptive and personalized persuasive systems. Creating successful persuasive applications requires a combination of design, social science, and technology. `StreamingBandit` contributes to the required technology by providing a platform that can be used to adapt persuasive technologies in real-time and at large scales. We first introduce the design philosophy of `StreamingBandit` using a running example and highlight how a large number of adaptive persuasive systems can be regarded as solutions to (contextual) multi-armed bandit problems: a type of problem that `StreamingBandit` was built to address. Subsequently, we detail several scenarios of the use of `StreamingBandit` to create adaptive persuasive systems and detail its future developments.

Researchers in the persuasive technology field have demonstrated the effectiveness and utility of persuasive applications in diverse domains such as health-care, energy reduction, and interactive marketing (see, e.g., Tuomas and Oinas-Kukkonen, 2009; Kaptein et al., 2012; Fogg, 2002). It is striking to see that applications and technologies that were researched by persuasive technology scholars half a decade ago are now commonplace: for example, devices that track physical activity of users and provide interactive feedback are commercially available on a large scale (e.g., Consolvo et al., 2008; Kaptein and van Halteren, 2012), and well-researched feedback methods – such as goal setting – have found numerous practical applications (Lacroix et al., 2008, 2009; Oinas-kukkonen and Harjumaa, 2009). As such, the field has made a noticeable mark on society.

However, there is an aspect of persuasive technologies that has long been advocated by scholars, but has by-and-large not left the research realm; this is the basic notion that persuasive technologies should "deliver the right message, at the right time, to the right user" (see, e.g., Hirsh et al., 2012; Oinas-kukkonen and Harjumaa, 2009; Kaptein et al., 2012). Although there is a common understanding that persuasive technologies *should* be made both adaptive and personalized, creating such systems is challenging. The successful development of adaptive persuasive systems requires a combination of design, social science, and technology. In this paper we contribute to the latter: we introduce `StreamingBandit`, a platform that supports the "logic and reasoning" of adaptive or personalized persuasive systems.

`StreamingBandit` is available open-source to those wishing to create adaptive persuasive systems. In this paper we detail the design rationale of the platform, document its core functionality, and present several use-cases.

Section 1 introduces a formalization of adaptive persuasive systems that is the starting point for `StreamingBandit` using a running example of an application to promote physical activity which we coin *RunSmart*. Section 2 introduces the platform and details how RunSmart can be implemented. In Section 3 we discuss a number of additional features of the platform, and in Section 4 we present additional use cases. Finally, we discuss the limitations of the current project, and encourage readers to contribute to the further development of the platform.

1. Formalizing adaptive persuasive systems

To detail the design choices made in `StreamingBandit` we first introduce the formalism that we have adopted to create the platform. We will use a running example of an adaptive application designed to promote physical activity coined RunSmart:

RunSmart is an application that encourages users to work out more. The system is composed of a smart-phone that can provide user feedback, and a (water resistant) accelerometer which allows the application to measure user activity. Every day the application sets a goal for the user. This goal is either a number of kilometers to run that day, or a number of kilometers to swim. Subsequently, the application measures the performed activities. RunSmart is adaptive based on the weather conditions: RunSmart will propose indoor swimming on rainy days, and outdoor running on sunny days. RunSmart is also personalized: the recommended goal is personal for each user. Since goals that are set too low are likely to lead to low activity levels, and goals that are set too high are likely demotivating and thus will also result in low activity levels, RunSmart tries to set a personal, challenging, goal.

RunSmart can be formalized such that the technical challenges of adapting the activity to the weather conditions and personalizing the goal can be solved quite easily using `StreamingBandit`. We introduce the following notation:

- An index of the interactions $t = 1, \dots, t = T$, where in our running example $t = 1$ is the time the first goal is set by RunSmart and $t = T$ the last (T is likely undefined at design time).
- The *context* $x_t \in \mathcal{X}_t$ where $\mathcal{X}$ is a set of variables describing the current state of the application. For RunSmart the relevant context vector x consists of only two variables ($x^{(1)}$ and $x^{(2)}$), of which one describes the weather ("rainy" or "sunny"), and one identifies the current user. Note that we use *super*scripts to identify elements of the context vector, and *sub*scripts to identify time points.
- The *action* $a_t \in \mathcal{A}_t$ where $\mathcal{A}$ is a set of possible actions the application can take. In RunSmart the action vector consists of two

variables, ($a^{(1)}$ and $a^{(2)}$); one denoting the activity ("running" or "swimming"), and one denoting the goal in kilometers.

- The *reward* r_t is a (function of the) measured response at that point in time. In our running example this is the activity of the user as measured using the accelerometer. For simplicity we assume that RunSmart directly measures the number of kilometers that was swam or ran.
- A *policy* $\Pi\colon x_1, \ldots, x_{t'}, a_1, \ldots, a_{t'-1}, r_1, \ldots, r_{t'-1} \to a_{t'}$, which is a mapping from all possible interactions (their contexts, actions, and rewards) up to some point in time $t = t'$ to the next action $a_{t'}$. In our example this is the "rule" that is used to select the next activity and goal (at time $t = t'$) given all the previous observations.

The above provides a formalization of the sequential choices made by RunSmart in such a way that these can be used directly in `StreamingBandit`. In short, `StreamingBandit`, provides a platform that allows users to implement different policies: different "rules" to assign new actions based on the historical interactions and the current context.

Note that the above formalization of the adaptive persuasive system in terms of context, actions, and reward, is known as a contextual multi-armed bandit problem (cMAB, or MAB for the simpler version without a context) (Berry and Fristedt, 1985; Whittle, 1980; Agrawal, 2014). The cMAB formalization is a reinforcement learning problem that is studied in mathematics and computer science as a model for decision making in uncertainty, where the values of the actions need to be learned sequentially (Sutton and Barto, 1998). In the MAB problem the experimenter needs to balance trying out actions to learn about their associated reward, and selecting those actions that she believes to lead to high rewards. In the cMAB problem the experimenter is, prior to making the decision, confronted with additional information, the context, which might influence the reward associated with certain actions but cannot be manipulated. The experimenter is looking for a policy Π which maximizes the cumulative reward $R(t) = \sum_{t=1}^{T} r_t | x_t, a_t$. A preferred policy can be conceived as the decision "rule" that selects actions given a context which produce the highest cumulated reward. For RunSmart we are looking for a policy that suggests goals (the actions), which give rise to the most active lifestyle of the user (the reward), given the specific user and the weather.

`StreamingBandit` formalizes the challenge of designing adaptive persuasive systems as a cMAB problem, and allows designers to implement a policy on a webserver that can be integrated in the persuasive application. To ensure the scalability of `StreamingBandit` we have made the design choice to split the implementation of a policy into two steps. To do so, we assume that at time $t = t'$, the (expected) effect of the choice $a_{t'}$ given context $x_{t'}$ on the reward $r_{t'}$ can be summarized into a model: $r = f(a, x; \theta_{t'})$. Given this assumption, we can identify the following two steps of a policy:

(1) The *summary* step: In each summary step $\theta_{t'-1}$ is updated by the new information $\{x_{t'}, a_{t'}, r_{t'}\}$. Thus, $\theta_{t'} = g(\theta_{t'-1}, x_{t'}, a_{t'}, r_{t'})$ where $g()$ is some update function. Effectively, all the prior data, $x_1, \ldots, x_t'$, $a_1, \ldots, a_{t'}$, $r_1, \ldots, r_{t'}$ are thus *summarized* into $\theta_{t'}$ and the model $f()$, which jointly is assumed to have a much lower dimension then the data itself. This choice for a summary at each time point makes that the computation time is bounded by the dimension of θ.[1]
(2) The *decision* step: In the decision step, the model $r = f(a, x_{t'}; \theta_{t'})$ is evaluated for the current context and the possible actions. Then, the recommended action at time t' is selected. Note that one could naively think that $a_{max} = argmax_a f(a, x_{t'}; \theta_{t'})$ would be the best action to choose. However, this ignores the uncertainty in both $f()$ and θ.

The above might sound a bit theoretical and detached from the design proces of adaptive persuasive systems. However, the summation step and decision step are easily related to our RunSmart example: in the *summation* step, RunSmart learns which goal leads to the highest reward based on the historical data. For example, a very simple model for RunSmart could be that the highest rewards is achieved when a goal that is 10% higher then the average exercise length of a user is suggested to that user. In this case the summary step merely updates the average exercise length of a user each time new data enters: for example, one specific user on average runs 5 km and swims 1 km. These *summaries* are the elements of θ for this user and are updated when newly measured activity data is received. Second, when a *decision* needs to be made, the weather is inspected (contained in x_t^1), the summary ($\theta_{t'}$) is retrieved, and a decision is made: If it is raining ($x_t^1 =$ "$rain$") the user is recommended to go for a 1.1 km long swim ($\bar{x}_{swim} * 1.1$). Thus, $a^1 =$ "$swim$", $a^2 = 1.1$. Note that both the model and the decision rule presented here are extremely simplistic; the model for the "best" goal should to be learned from the data. Also, the uncertainty in the estimate of the average sport lengths need to be incorporated, and integrated in the decision rule.

2. Introducing `StreamingBandit`

In this section we introduce `StreamingBandit` and demonstrate how it can be used to implement the adaptation and personalization of RunSmart. Figure 2 presents an overview of the platform and it's main functionality. `StreamingBandit` is a python 3 application that runs a Tornado webserver (see `http://www.tornadoweb.org/en/stable/`) and which discloses a REST API that facilitates the implementation of the *summary* and *decision* steps as described above. The two main REST calls are:

[1]Note that this forces one to implement an online policy (Michalak et al., 2012). While perhaps not intuitively clear, this forces users of `StreamingBandit` to write summary schemes that are computationally bounded and scalable.

- The *decision* call:

```
.../HOST/EXPID/getaction.json?key=EXPKEY&context={}
```

  where `.../HOST` is the address where the `StreamingBandit` server is running, EXPID and EXPKEY are the ID and key of the current application (see Section 2.1.1), and the variable context contains a JSON object encoding the context x_t (e.g., {`"weather":"rainy", "userid":12`}). The call returns a JSON formatted object containing the selected action given the policy which depends on the current summarized data θ_t.[2]
- The *summarize* call:

```
.../HOST/EXPID/setreward.json?key=EXPKEY&context={}
    &action={}&reward={}
```

  where the context x_t, action a_t and reward r_t at a point in time are used to update θ_t.

`StreamingBandit` allows users to create a new "experiment" to enable the above two calls, and allows users to write custom `python 3` scripts that implement the *summary* and *decision* steps.

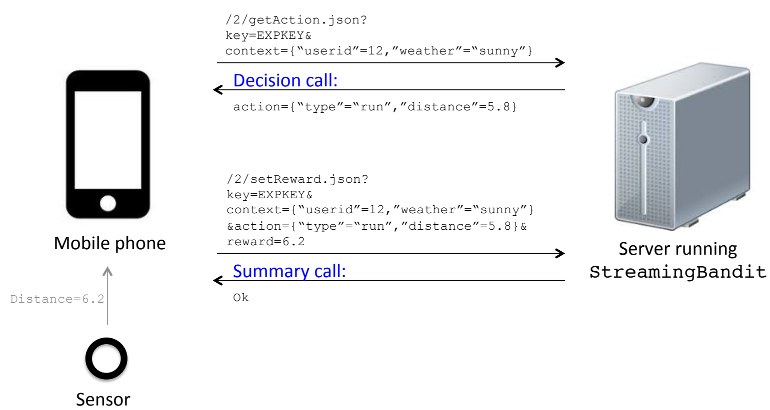

FIGURE 1. Schematic overview of the core functionality of `StreamingBandit`, including two example REST API calls for the RunSmart application.

2.1. **Configuring `StreamingBandit`.** Configuring `StreamingBandit` consists of three steps; first, one creates a new experiment which initializes up the associated REST calls. Second, one implements the *summary* step logic in a custom `python` script which can be done using the web-based front-end of `StreamingBandit`. Finally, one implements the *decision* step. We explain each in detail below.

[2]JSON is selected as the format for passing data from and to the server because of its omnipresence in web based applications

2.1.1. *Step 1: Setup an experiment.* Step one is setting up a new experiment. Figure 2.1.1 presents a number of screenshots of the `StreamingBandit` front-end that can be used to setup new experiments by navigating to

```
.../HOST/management.html
```

The user is asked to provide a name for the experiment, and to provide the code for the summary and decision steps (see explanation below). After creating the experiment, it receives an experiment ID and key that need to be used in the subsequent REST calls. After creating the adaptive goal setting experiment it received `id=1` and `key=36207d46df` (see lower right panel), now enabling calls to (e.g.,)

```
.../HOST/1/getAction.json?key=36207d46df&context={}
```

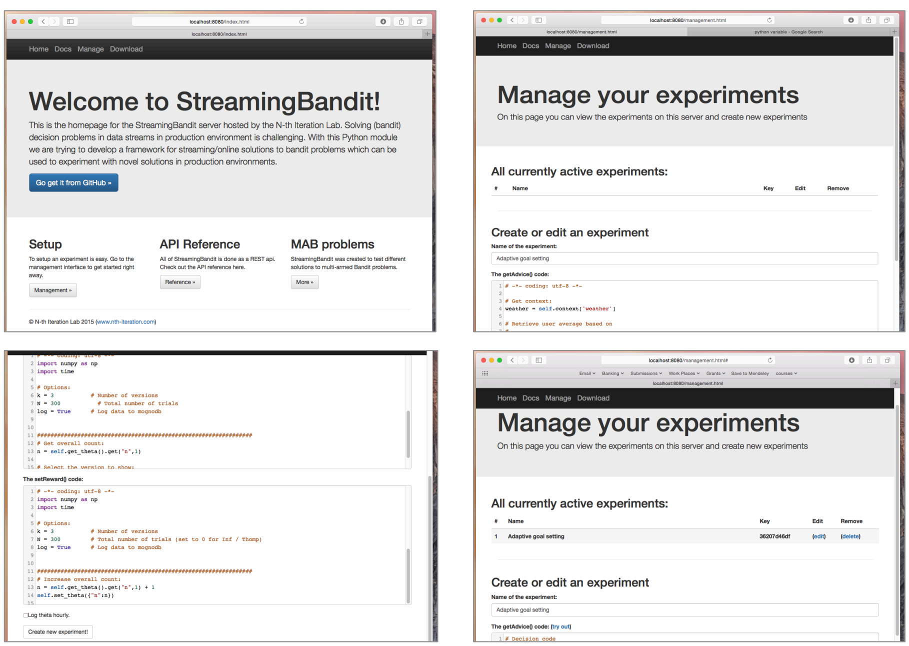

FIGURE 2. Screen shots of the front-end of `StreamingBandit`. Upper left is the welcome page (`http://HOST`). After clicking "management" one can specify a name for the experiment (upper right), and specify the code for the summary and decision steps (lower left). After the experiment is created it receives an id and key (lower right).

2.1.2. *Step 2: Implement the summary step.* Next, the actual logic needs to be implemented. For illustration purposes we continue with the Run-Smart application, and focus on a *summary* step which updates the average distance traveled for each user independently for both swimming and running. We enable this functionality by writing a custom script which, given a context, an action, and a reward, computes the averages. Consider the call:

```
.../HOST/1/setreward.json?key=36207d46df&
    context={"weather":"sunny", "userid":12}&
    action={"type":run, "distance":6}&reward={"km":8}
```

which denotes that user number 12 on a sunny day received the goal to run for 6km (the action a_t), and ended up running 8 kilometer (the reward, r_t). We now update the average run length of user 12 that is stored in θ to incorporate the new information. To do so, we add the following summary code:

```
1 import libs.base as base
2 key = "weather-uid"
3 value = self.context["weather"] + str(self.context["userid"])
4 average = base.Mean(self.get_theta(key=key, value=value))
5 average.update(self.reward["km"])
6 self.set_theta(average, key=key, value=value)
```

Any standard `python3` code can be used to define the summary step. However in the above code snippet we make use of a number of the functionalities implemented in `StreamingBandit`; these features are documented in detail in the manual that is available at `http://mkaptein.github.io/streamingbandit/`.

The code above first imports a separate toolkit from `libs.base` that is shipped with `StreamingBandit`. Our use of toolkits enable third-party developers to add functionality to `StreamingBandit`. Next, the current average for the current user and the current weather (both encoded in the `context` object, see lines 2-3) are retrieved in line 4, and it is casted as a "base.Mean" object. This object is updated with the number of kilometers ran (the reward) in line 3, and finally the new average for this user and this weather is stored.

2.1.3. *Step 3: Implement the decision step.* For the *decision* step the code that is used is again standard `python3` code, possibly augmented with `StreamingBandit` functionality. For our example application the REST call to retrieve a new goal is:

```
.../HOST/1/getAction.json?key=36207d46df&
   context={"weather":"sunny", "userid":12}
```

and the following code implements the response logic:

```
import libs.base as base
key = "weather-uid"
value = self.context["weather"] + self.context["userid"]
average = base.Mean(self.get_theta(key=key, value=value))
self.action['distance'] = average.get_value() * 1.1
if self.action['distance'] == 0:
    self.action['distance'] = 1
```

```
8 self.action['type'] = "run" if
9     self.context["weather"] == "sunny" else "swim"
```

Here, line 4 again retrieves the average for the current user and context (as in the previous step), and line 5 computes the suggestion by adding 10% to this average. Line 6-7 set an initial goal of 1 kilometer if no data is present. Finally, Line 8-9 add, based on the context, the suggested activity type. The `self.action` object is automatically returned and hence the server response will be a JSON object containing the variables "distance" and "type".

Step 1–3 implement the server-side of the development of adaptive persuasive systems. Once the experiment is setup the two REST API calls are available. The smart phone application can now be integrated. For many smart-phone and web applications implementing the two REST calls and handling the responses will be straightforward since HTTP REST and JSON are common standards.

Note that the three step process presented here implements a very rudimentary version of RunSmart's logic: the weather is tied one-to-one to the activity, the data of each user is treated independently, the goal setting model (adding 10%) is fixed, and no attention is given to the uncertainty in any of the obtained estimates. In real applications one would likely use a much more sophisticated (e.g., hierarchical) model to model the relationship between the actions, the context, and the rewards, and use some method of balancing exploration and exploitation (e.g., Thompson sampling, Chapelle and Li, 2011). One of the advantages of `StreamingBandit` however is the fact that once setup, one can easily extend and improve the logic of the adaptation.

3. Additional features

In this section we briefly describe a number of features of `StreamingBandit` that might be of interest for designers of persuasive systems. First, `StreamingBandit` comes with a number of different libraries (such as `libs.base` mentioned above) to enable streaming processing of data. The `libs.base` toolkit allows researchers to easily update counts, means, proportions, variances, and co-variances in data streams. Similarly, the `libs.lm` library allows fitting of linear regression models in data streams. The `libs.thompson` library implements Thompson sampling (e.g., Chapelle and Li, 2011) for the Bernoulli bandit problem. Currently we are working on developing more toolkits, and invite others to contribute toolkits together with documented examples. Next to the toolkits, `StreamingBandit` comes with a number of default implementations for simple policies such as ϵ-first (the AB test), Thompson sampling, and a number of others. These are available through the management console and can easily be expanded (see also Section 4).

`StreamingBandit` also allows logging of all the calls that are made and all the data that is collected. Within the custom summary and decision steps explicit calls to `self.log_data({})` can be made to store JSON objects encoding (e.g.), the current calls, the time, etc. This allows developers of adaptive persuasive systems to run post-hoc analysis on the outcomes of an evaluation. The collected logs, as well as the current state of θ are available through separate REST calls and can thus also be integrated into applications. Similarly, the management front-end, as depicted in Figure 2.1.1, is fully detached from the core and its functionality is enabled through a number of separate REST calls: hence developers can easily develop their own front-end to setup experiments.

We have tried to make `StreamingBandit` both scalable and secure: the management console can easily be protected using signed cookies, and the REST calls to the the core *summary* and *decision* steps are protected using the experiment id and key combination. Scalability is ensured by a) using state-of-the art technologies in the development of `StreamingBandit` such as Tornado (as a base for the webserver), and Redis (an extremely fast, in-memory data-base system), and b) by forcing, by design, the use of online (streaming) analysis.

Finally, `StreamingBandit` allows "nested" experiments: hence, the summary or decision steps of an experiment can call, using `self.run_experiment(id)`, the summary or decision code of *another* experiment. This extremely powerful feature allows one to implement complex logic comparing – using e.g., an AB test (see Section 4.1 – multiple methods of adaptation or personalization). This latter functionality for example would enable designers of the RunSmart app to develop multiple goal setting algorithms, and compare the effectiveness of each within a single experiment. The nesting of experiments makes `StreamingBandit` a flexible platform to test, evaluate, and iterate different adaptation and personalization attempts.

4. Possible scenarios

In this section we describe a number of possible uses of `StreamingBandit`. This section aims to highlight the versatility `StreamingBandit` has to offer for designers of adaptive persuasive systems; it does not aim to give a full overview of the implementation of these scenarios.

4.1. **AB testing.**

A simple application of `StreamingBandit` is AB testing: comparing two versions of an interface in a randomized controlled experiment. In its simplest case there is no context (thus, this is a MAB problem, not cMAB), there are two possible actions $a_t \in \{A, B\}$, and the reward is a click on a webpage ($r_t \in \{0, 1\}$). The summarize step of such an AB test can be implemented as follows:

```
import libs.base as base
prop = base.Proportion(self.get_theta(key="version",
    value=self.action["version"]))
prop.update(self.reward["click"])
self.set_theta(prop, key="version", value=self.action["version"])
```

This code updates and stores the click proportion of the different versions (the actions) of the application. The decision step is given by:

```
import libs.base as base
propl = base.List(self.get_theta(key="version"),
  base.Proportion, ["A", "B"])
if propl.count() > 1000:
  self.action["version"] = propl.max()
else
  self.action["version"] = propl.random()
```

Which implements that up to 1000 interactions ($t = 1000$) randomly version A and B are suggested. After 1000 interactions, the version with the highest click through proportion is suggested. Note that not providing a value in the `self.get_theta` call results in a list of the data for all possible values.

4.2. **Dynamic AB testing.** The AB test presented above, while commonplace, ignores the uncertainty in the estimated proportions. Instead of adopting a fixed time exploration period (the experiment, $t < 1000$), and then moving to an exploitation period (selecting the version that performed best during the trial), one can also balance exploration and exploitation in a more refined fashion.[3] This is also coined *dynamic* AB testing, and one selects a version, "A" or "B", proportional to the belief that that version has the highest reward (this is a policy known as Thompson sampling, see Thompson, 1933). To implement a dynamic AB test the summary step remains as above, and the decision step is changed to:

```
import libs.thompson as thmp
propl = thmp.BBThompsonList(self.get_theta(key="version"),
   Proportion, ["A", "B"])
self.action["version"] = propl.thompson()
```

This example highlights that the summary step and the decision step are loosely coupled, and that sometimes new policies can be created by combining existing summary and decision steps.

4.3. **Learning a linear model.** The RunSmart example application as discussed in Sections 1 & 2 can easily be altered by changing the goal-setting method. One way of doing so would be to remove the assumption that an increase of 10% over the current average distance is optimal and replace it by a goal that is learned from the rewards. We could for example define δ as the difference between the average distance $\bar{x}$ and the set goal g, and learn how δ relates to the actual reward using a simple linear model $r = \beta_0 + \delta\beta_1 + \delta^2\beta_2 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The value of the coefficients, $\beta_0, \ldots, \beta_2$, of this simple model can be learned from the data. This can be done using the following code in the summary step:

[3]Space limitations in this paper do not allow us to dig into the details of the exploration-exploitation trade-off, or the possible methods to address this issue. For more information we recommend (Macready and Wolpert, 1998).

```
import libs.base as base
import libs.lm as lm
key = "weather-uid"
value = self.context["weather"] + str(self.context["userid"])
model = lm.Model(self.get_theta(key=key, value=value))
mean = base.Mean(self.get_theta(name="mean", key=key, value=value))
d = mean - self.action["km"]
model.update(self.reward["km"], [d, d^2])
mean.update(self.reward["km"])
self.set_theta(model, key=key, value=value)
self.set_theta(average, name="mean", key=key, value=value)
```

The code above will fit the linear model in a data stream, and the coefficients will be available in the decision step using `model.get_coefs()` and can be used to determine the optimal goal.

It has to be noted that the examples provided above are relatively simple: they use small or no context, and simple updating models. This somewhat obscures the true potential of `StreamingBandit` to select actions in very complex settings at very large scales. However, we hope that the above examples introduced the main ideas behind `StreamingBandit` and encourages interested readers to develop their own policies.

5. Limitations and Conclusions

`StreamingBandit` is still work in progress: Although we have recently released the first stable version of the application on GitHub, we are still developing additional toolkits, examples, and documentation. Also, we are currently using `StreamingBandit` in the evaluation of adaptive persuasive technologies; we hope to be able to report on actual field trials powered by `StreamingBandit` in the near future. However, we think the current version is mature enough to share with the Persuasive Technology community, and to encourage others to use the application. We are actively seeking feedback to improve the application and maximize its use.

We are currently aware of a number of limitations: first, the application is currently run and tested only on a single core. While parallelization for large scale is easy at the level of experiments, this will be harder within experiments; this is an obvious next step in ensuring scalability. Also, the toolkits could be greatly extended and improved; for example, many methods to address exploration-exploitation trade-off in various settings have been developed in the machine learning and statistical literature and could be included.

In this paper we have briefly introduced `StreamingBandit` and presented relatively simple examples. However, we hope the current expose is sufficient to a) communicate the design philosophy behind `StreamingBandit`, and b) raise the interest of those developing adaptive or personalized persuasive systems wishing to easily implement their adaptation logic: this is exactly where

`StreamingBandit` can be of help. The latest version of `StreamingBandit` can be downloaded from `https://github.com/MKaptein/streamingbandit`.

References

Agrawal, S. (2014). Thompson Sampling for Contextual Bandits with Linear Payoffs.

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Springer.

Chapelle, O. and Li, L. (2011). An Empirical Evaluation of Thompson Sampling. In Shawe-Taylor, J., Zemel, R. S., Bartlett, P., Pereira, F. C. N., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems*, pages 2249—-2257.

Consolvo, S., Libby, R., Smith, I., Landay, J. A., McDonald, D. W., Toscos, T., Chen, M. Y., Froehlich, J., Harrison, B., Klasnja, P., LaMarca, A., and LeGrand, L. (2008). Activity sensing in the wild. In *Proceeding of the twenty-sixth annual CHI conference on Human factors in computing systems - CHI '08*, page 1797, New York, New York, USA. ACM Press.

Fogg, B. J. (2002). *Persuasive Technology: Using Computers to Change What We Think and Do*. Morgan Kaufmann.

Hirsh, J. B., Kang, S. K., and Bodenhausen, G. V. (2012). Personalized Persuasion: Tailoring Persuasive Appeals to Recipients' Personality Traits. *Psychological Science*, (April).

Kaptein, M. C., de Ruyter, B., Markopoulos, P., and Aarts, E. (2012). Tailored Persuasive Text Messages to Reduce Snacking. *Transactions on Interactive Intelligent Systems*, 2(2):10—-35.

Kaptein, M. C. and van Halteren, A. (2012). Adaptive Persuasive Messaging to Increase Service Retention. *Journal of Personal and Ubiquitous Computing*, 17(6):1173–1185.

Lacroix, J., Saini, P., and Goris, A. (2009). Understanding user cognitions to guide the tailoring of persuasive technology-based physical activity interventions. In Chatterjee, S. and Dev, P., editors, *Persuasive Technology, Fourth International Conference, PERSUASIVE 2009*, volume 350 of *ACM International Conference Proceeding Series*, page 9, Claremont, California, USA. ACM.

Lacroix, J., Saini, P., and Holmes, R. (2008). The relationship between goal difficulty and performance in the context of a physical activity intervention program. In *Proceedings of the 10th international conference on Human computer interaction with mobile devices and services - MobileHCI '08*, page 415, New York, New York, USA. ACM Press.

Macready, W. G. and Wolpert, D. H. (1998). Bandit problems and the exploration/exploitation tradeoff. *IEEE Transactions on Evolutionary Computation*, 2(1):2–22.

Michalak, S., DuBois, A., DuBois, D., Wiel, S. V., and Hogden, J. (2012). Developing Systems for Real-Time Streaming Analysis. *Journal of Computational and Graphical Statistics*, 21(3):561–580.

Oinas-kukkonen, H. and Harjumaa, M. (2009). Persuasive Systems Design : Key Issues , Process Model , and System Features. *Design*, 24.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to reinforcement learning*. MIT Press.

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.

Tuomas, L. and Oinas-Kukkonen, H. (2009). The persuasiveness of Web-based alcohol interventions. In *9th IFIP Conference on e-Business, e-Services, and e-Society*, pages 164–176.

Whittle, P. (1980). Multi-armed bandits and the Gittins index. *Journal of the Royal Statistical Society Series B Methodological*, 42(2):143–149.

Tilburg University, Tilburg, the Netherlands. Statistics and Research Methods.

E-mail address: m.c.kaptein@uvt.nl

Radboud University, Nijmegen, the Netherlands. Artificial Intelligence.