An Empirical Evaluation of Context Aware Clustering of Bandits using Thompson Sampling

NICOLÒ CAMPOLONGO
An Empirical Evaluation of Context Aware Clustering of Bandits using Thompson Sampling

Nicolo’ Campolongo

Master’s Programme: Machine Learning

Supervisor: Alexandre Proutiere, KTH
Examiner: Danica Kragic, KTH
Principal: Claudio Gentile, University of Insubria/INRIA
Abstract

Stochastic bandit algorithms are increasingly being used in the domain of recommender systems, when the environment is very dynamic and the items to recommend are frequently changing over time. While traditional approaches consider a single bandit instance which assumes all users to be equal, recent developments in the literature showed that the quality of recommendations can be improved when individual bandit instances for different users are considered and clustering techniques are used.

In this work we develop an algorithm which clusters users based on the context at disposal using a Bayesian framework called Thompson Sampling, opposed to a similar algorithm called CAB recently presented at ICML 2017 (International Conference on Machine Learning), which uses a deterministic strategy. We show extensively through experiments on synthetic and real world data that the performance of our algorithm is in line with its deterministic version CAB when it comes to the quality of recommendations. On the other hand, our approach is relatively faster when considering running time.
Acknowledgements

This work was performed at the department of Computer Science at the university of Insubria, supported by the Erasmus+ scholarship.

I would like to thank my supervisor at the university of Insubria, prof. Claudio Gentile for his constant support during the time I spent there and for all the fruitful discussions we had.

Thank you to my supervisor at KTH, Alexandre Proutiere, for introducing me to the world of bandits during my master’s. His course on Markov Decision Processes was one of the most interesting I attended.

Thank you Danica Kragic for your always prompt responses and examining this thesis.

Thank you Ann Bengtsson for making sure that this project complied with all administrative aspects.

Finally I would like to thank my girlfriend, Anisa, for proofreading this thesis.
# Contents

1 Introduction .......................................................... 1
   1.1 Bandits and Recommender Systems ............................ 2
   1.2 Thesis Contribution ........................................... 4
   1.3 Limitations .................................................... 4
   1.4 Thesis outline ................................................ 4

2 Background ........................................................... 6
   2.1 Definitions .................................................... 6
      2.1.1 Regret .................................................. 7
      2.1.2 A fundamental result .................................. 7
   2.2 Algorithms .................................................... 8
      2.2.1 $\epsilon$-greedy ........................................ 9
      2.2.2 Upper Confidence Bound ................................. 9
      2.2.3 Thompson Sampling ...................................... 10
   2.3 Contextual bandits ............................................. 11
   2.4 Stochastic linear bandits ..................................... 12
      2.4.1 LinUCB / OFUL .......................................... 12
      2.4.2 Linear Thompson Sampling ............................... 14
   2.5 Bandits on graphs ............................................. 15
      2.5.1 Context-Dependent Clustering of Bandits ............... 16

3 Method ............................................................... 19
   3.1 ThompCAB ..................................................... 19
      3.1.1 Setting ................................................ 19
      3.1.2 Algorithm description .................................. 19
      3.1.3 Keeping count of the number of updates ................ 21
      3.1.4 Implementation ......................................... 21
      3.1.5 Efficiency ............................................. 23
   3.2 Algorithms ..................................................... 23
   3.3 Choice of the parameters .................................... 23
   3.4 Datasets ....................................................... 24
   3.5 Artificial Data ............................................... 24
   3.6 Avazu .......................................................... 25
      3.6.1 Feature Hashing ........................................ 26
      3.6.2 Feature conjunctions ................................... 28
      3.6.3 Policy evaluation ...................................... 29
   3.7 Software and Platform ........................................ 29
   3.8 Evaluation ..................................................... 30
3.8.1 Regret ........................................ 30
3.8.2 Running Time .................................. 31

4 Results .............................................. 32
  4.1 Experiment 1: studying the gap parameter $\gamma$ .............. 32
  4.2 Experiment 2: varying the number of interactions per user .... 35
  4.3 Experiment 3: varying the number of users .................... 36
  4.4 Experiment 4: bandits on real world data .................... 38

5 Conclusions ....................................... 42
  5.1 Discussion and future work .................................... 42
  5.2 A view on ethics and society .................................. 43
List of Figures

1.1 One-armed bandit machine. ................................. 1

3.1 Latent classes used for generating artificial users. .............. 25
3.2 Users distribution: on the x-axis the number of times a user interacted with the system divided into bins, on the y-axis the log count of users falling into each bin. ............................................. 27

4.1 First variant of the algorithm used in the first experiment. Each algorithm is run 10 times and the average is plotted. We do not show the standard deviation by plotting confidence bounds in order to avoid clutter. ......................................................... 33
4.2 Second variant of the algorithm used in the first experiment. Each algorithm is run 10 times and the average is plotted. We do not show the standard deviation by plotting confidence bounds in order to avoid clutter. ......................................................... 34
4.3 Averaged cumulative regret plotted in the interval $T_u \in \{10, \ldots, 100\}$. In particular, randomized algorithms are averaged on 10 runs and error bars are reported showing the standard deviation. .............. 35
4.4 Average regret per user when varying the number of users. ....... 36
4.5 Running time of the algorithms: on the x-axis the number of users considered, on the y-axis the running time in seconds. .............. 37
4.6 Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 1000 users. In this case features conjunctions are not used. ......................................................... 39
4.7 Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 1000 users. In this case features conjunctions are used. ......................................................... 40
4.8 Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 10000 users. ......................................................... 41
List of Tables

3.1 Unique values features can take. ................................................. 27
3.2 Description of the 3 final datasets built from Avazu data. ”d” is the dimension of the contexts, ”k” the number of arms and ”conj” denotes the presence of features conjunctions. ................................. 30

4.1 Table describing the results of the first experiment when using the first variant of the Thompson CAB algorithm. ................................. 32
4.2 Table describing the results of the first experiment when using the second variant of the Thompson CAB algorithm with different parameters. ................................................................. 33
4.3 Table describing the running time of the algorithms in seconds when varying the number of users. .................................................. 37
4.4 Results on the dataset built with the most frequent 1000 users. In this case features conjunctions are not used. ................................. 39
4.5 Results on the dataset built with the most frequent 1000 users. In this case features conjunctions are used. ................................................ 40
4.6 Results on the dataset built with the most frequent 10000 users. In this case the probability of sampling users when building the neighbourhoods was $p = 0.1$ for both CAB and ThompCAB. .................. 41
Chapter 1

Introduction

Multi-armed bandit formulation [Bubeck and Cesa-Bianchi, 2012] is a general framework to deal with sequential decision making in the face of uncertainty. In the standard multi-armed bandit scenario we have an agent (the learner) which interacts with the environment in rounds. In particular, every round it chooses an action (also called arm) and gets a feedback from it (reward) which measures how ”good” the choice was.

For rounds \( t = 1, 2, 3, ... \)

1. The learner chooses an action from a set of available actions.
2. The environment sends back its response, in form of a reward.

The goal of the learner is to maximize the sum of rewards it receives, or in other words to minimize its regret with respect to the best action. In general, bandit optimization is part of the online learning scenario, where data becomes available in a sequential manner. Thus, the learning process does not happen all at once by processing the entire dataset at disposal. On the contrary, online algorithms need to dynamically adapt to new emergent patterns in the data.

Why multi-armed bandits? Back in the ’50s a group of scientists studying human behaviour conducted experiments by making people pull one of the two arms in a ”two-armed bandit” machine, with each arm having a random payoff according to some distribution unknown to the users. One-armed bandit machine is an old name for a lever operated slot machine (”bandit” since it steals the money!), such as the one depicted in Figure 1.1.

The bandit framework was firstly investigated by William R. Thompson back in 1933 [Thompson, 1933] in a paper describing strategies to adopt in clinical trials, with the goal of designing a treatment selection scheme in order to maximize the number of patients cured after

Figure 1.1: One-armed bandit machine.
Here we have two available treatments (red or blue pill) with unknown rewards: "Live" (L) or "Die" (D). After administrating the treatment to a patient, we observe whether he survives or dies. In this case, the red pill looks more promising compared to the blue one. However, would it be convenient to only use the red pill for future treatments? Maybe we just had some bad luck with the blue pill, it might be worth trying it again.

This situation captures the fundamental dilemma behind bandit optimization: should one explore an action that looks less promising, or should one exploit the currently best valuable action? How to maintain the balance between exploration and exploitation is at the heart of bandit problems.

Applications

Modern applications of these algorithms include different scenarios as explained below.

- **News**: when a user visits a news website, the website shows different headers which can be clicked. The headers represent the arms and the objective is to maximize the number of clicks. In this case, the reward can either be 0 (no click) or 1 (click).

- **Ad placement**: website advertising is a major source of revenue for big companies such as Google and Facebook. When a user visits a webpage, there is a learning algorithm which selects one of the many ads it can display. In this case, the reward can be considered the amount of money we get if the ad gets clicked while the arms are the different ads.

- **Rate adaptation**: transmitters in wireless systems can adapt the coding and modulation scheme to the radio channel conditions. This mechanism is called rate adaptation. In this scenario, the arms are represented by a set of transmission rates and the algorithm’s objective is to maximize the product of the rate and the success transmission probability at this rate.

1.1 Bandits and Recommender Systems

Recommender systems are information systems whose goal is to predict the rating a user would give to an item in order to provide the best possible recommendation to such a user. Nowadays they are used in multiple domains, ranging from movies or music recommendations (i.e. Netflix and Spotify) to news (Yahoo! News), search queries (Google) and products to be sold (Amazon).

When user features are available (such as historical activities or demographic information), the main approach in recommender systems is the collaborative filtering
technique: based on ratings given to some items by users, the system aims to infer similarities within users and items. The idea behind this is that people with similar tastes can get similar recommendations (user-based collaborative filtering [Park et al., 2006]). On the other hand, similar items will be rated in the same way by users (items-based collaborative filtering [Park et al., 2006]). In practice, collaborative filtering recommendations are performed through matrix factorization [Rennie and Srebro, 2005], a mathematical technique made popular by the Netflix challenge [Bennett and Lanning, 2007]. The idea behind matrix factorization is to learn a lower dimensional space on latent features underlying the interactions between users and items.

Matrix factorization techniques work well in practice, especially when the environment considered is static, i.e. music or movies recommendations where the pools of content is not rapidly changing over time. On the other hand, there are other domains where the environment is much more dynamic and fast-paced, making it fundamental to quickly identify interesting content for the users. For example, in news recommendations or ad placement the pool of content undergoes frequent insertions and deletions. Moreover, part of the users could be completely new to the website visited, which thus has no previous records of the user interests. This issue is known as the cold-start [Park et al., 2006] problem. In such scenarios, the goal of the system is to maximize user satisfaction over time while gathering information about matches between user interests and different content. Thus, the problem can be perfectly cast into the exploration/exploitation trade-off typical of multi-armed bandit algorithms.

During the last few years, multi-armed bandit algorithms gained success in recommender systems. When using bandits, we have two opposite approaches: the first is to discard differences between users and use one single bandit instance for all of them. More formally, we assume each user is drawn independently from a fixed distribution over users, so that in each round the reward depends only on the recommended item. However, when we deal with a lot of different users it may be difficult for a single bandit to learn a good model since different group of people may have different interests. The opposite alternative is to build a fully personalized system where each user has his own bandit instance, independent from all the other users.

In addition, recent advances show promising results when considering a graph structure to model users or items (see [Valko, 2016] for a survey on the subject). When the users are represented as nodes in a graph, we can exploit structures in such a graph (e.g. connections, clusters, ecc.) in order to provide better recommendations. On the other hand, bandit algorithms based on graph structures have to deal with the problem that the underlying graph structure is often not available in advance and must be learned on the fly. In order to circumvent this issue, in [Gentile et al., 2017] an algorithm is developed, called CAB (Context Aware clustering of Bandits). It does not require any knowledge of the underlying graph, where users can be added or discarded on the fly. This algorithm clusters users into different groups and gives recommendations in a collaborative way, i.e. the opinion of all the users in a cluster is considered.

The ideas contained in [Gentile et al., 2017] will be the starting point of the thesis project. In particular, we will consider a stochastic scenario, e.g. we assume the rewards being sampled from a probability distribution. In this setting, probably the most famous strategies belong to the family of Upper Confidence Bound (UCB)
algorithms, first described in [Agrawal, 1995] and [Auer et al., 2002a]. This is a class of deterministic algorithms whose goal is to compute an index for each arm and play the arm with the highest index, in a way that each arm is always explored enough. The algorithm described in [Gentile et al., 2017] is also based on a UCB strategy, too. On the other hand, in recent years bandit algorithms based on a Bayesian strategy called Thompson sampling have rapidly gained popularity. The reason of their success can be explained by their better performances in empirical applications, as shown for example in [Chapelle and Li, 2011] or [Scott, 2010]. For this reason, recent studies (see [Agrawal and Goyal, 2013b], [Agrawal and Goyal, 2013a]) investigated these algorithms from a theoretical point of view in order to provide theoretical guarantees and justify their empirical success. Finally, when considering the running time, Thompson sampling algorithms often offer more scalable solutions compared to UCB ones, as shown in [Kocak et al., 2014] or [S. Vaswani, 2017].

1.2 Thesis Contribution

Our objective in this work is to provide a Thompson sampling version of the CAB algorithm described in [Gentile et al., 2017] and give an empirical evaluation of its behaviour testing it on both synthetic and real-world data. In particular, we will compare its performance to CAB itself and other known algorithms in the bandit setting, such as LinUCB [Li et al., 2010] and Linear Thompson Sampling [Agrawal and Goyal, 2013b]. We will try to answer the following questions:

- How does the Thompson sampling version of CAB compare in terms of regret?
- How does the Thompson sampling version of CAB compare in terms of running time?

1.3 Limitations

Since the nature of this work is empirical, we will not conduct a theoretical analysis of the algorithm, which would be necessary in order to provide theoretical guarantees on the order of the regret. Also, we will not use very big datasets because of the large amount of time required to run the algorithm we develop, as shown later in the report.

1.4 Thesis outline

The thesis is structured in the following way: in chapter 2 we describe the relevant theory on stochastic bandit algorithms, introducing the common definitions and describing popular strategies adopted, such as Upper Confidence Bound schemes and Thompson sampling. Then, we review the related work on bandit for recommender systems. In particular, we first describe recent work similar to ours, i.e. trying to cluster users and later analyze the closest work to this thesis project, which is the CAB algorithm described in [Gentile et al., 2017].

In chapter 3 we describe our algorithm and its implementation. In particular, we describe the data structures we used and how we dealt with the problem of
scalability when the number of users is large. Then, we describe the setup for our experiments both on synthetic and real-world data.

In chapter 4 we present results on experiments that we ran. We illustrate our results through figures and tables.

In chapter 5 we give an overview on our approach, problems we encountered and finally suggestions for future work.
Chapter 2

Background

This chapter is an introduction to theory of stochastic multi-armed bandit problems. We first describe the terminology used within this field and then the most common algorithms in the literature. Finally we describe the CAB algorithm proposed in [Gentile et al., 2017], which is closely related to the one that we will develop and describe in section 3.1.

2.1 Definitions

We can have three different bandit scenarios: stochastic, adversarial and Markovian ([Bubeck and Cesa-Bianchi, 2012]). In this work, we are going to focus on stochastic bandits.

**Stochastic bandits** is an abbreviation for stochastic independent identically distributed (iid) bandits. For every action, the corresponding rewards are independent but all generated from the same distribution. Formally, the stochastic multi-armed bandit problem ([Auer et al., 2002a]) is defined as follows: an environment is given by $K$ distributions over the reals, $P_1, ..., P_K$ associated with a set $\mathcal{A} = \{a_1, a_2, ..., a_K\}$ of available actions (e.g. arms). The learner and the environment interact sequentially. The feedback that the environment gives to the learner at each time step is called **reward**. Usually, a positive reward represents a positive feedback, such as earning a sum, while a negative reward represents a negative feedback, such as a loss or a failure.

Let $h_t = (a_t, X_t)$ be the history for round $t$, where $a_t$ is the chosen arm and $X_t$ is the reward. We have:

For rounds $t = 1, 2, 3, ...$

1. Based on the current history $\mathcal{H}_{t-1} = \{h_1, h_2, \ldots, h_{t-1}\}$, the learner chooses an action $a_t$ from the set $\mathcal{A} = \{a_1, a_2, ..., a_K\}$ of available actions.

2. The environment sends back its response, in form of a reward value $X_t \in \mathbb{R}$, whose distribution is $P_{a_t}$ (e.g. $X_t \sim P_{a_t}$).

The learner’s objective is to maximize its total reward, $S_T = \sum_{t=1}^{T} X_t$. 

6
2.1.1 Regret

To study the performances of bandit algorithms, another metric is often used: the 
\textbf{regret}. If we denote the best action by \( a^* \), then the regret of the learner relative 
to action \( a^* \) is the difference between the total reward gained when \( a^* \) is used for 
all the rounds and the total reward gained by the learner according to its chosen 
actions.

Formally, we denote the expected reward of the \( k \)-th arm by \( \mu_k = \int_{-\infty}^{\infty} x P_k(x)dx \).
The expected reward of the best arm is denoted by \( \mu^* = \max_k \mu_k \). Then, the \textbf{gap} 
of the \( k \)-th arm for one single round is defined as \( \Delta_k = \mu^* - \mu_k \). If we have \( n \) rounds 
on total, then \( n_k = \sum_{t=1}^{T} I\{a_t = k\} \) is the number of times the \( k \)-th arm was chosen 
by the learner in \( T \) rounds. In general \( n_k \) is a random quantity since in each round \( t \) it 
depends on \( a_t \), which in turn depends on the previous random rewards observed.
The overall regret can then be written as \( R_T = T \mu^* - \mathbb{E}[\sum_{t=1}^{T} X_t] \).

\textbf{Lemma 2.1.1. Regret decomposition}: \( R_T = \sum_{k=1}^{K} \Delta_k \mathbb{E}[n_k] \)

\textbf{Proof.} We have that \( S_n = \sum_t X_t = \sum_t \sum_k X_t I\{a_t = k\} \). Then:

\[
R_n = T \mu^* - \mathbb{E}[S_n] \\
= \sum_{k=1}^{K} \sum_{t=1}^{T} \mathbb{E}[(\mu^* - X_t) I\{a_t = k\}] \\
= \sum_{k=1}^{K} \sum_{t=1}^{T} \mathbb{E}[(\mu^* - X_t) I\{a_t = k\}|a_t] \\
= \sum_{k=1}^{K} \sum_{t=1}^{T} I\{a_t = k\} \mathbb{E}[(\mu^* - X_t)|a_t] \\
= \sum_{k=1}^{K} \sum_{t=1}^{T} I\{a_t = k\} (\mu^* - \mu_k) \\
= \sum_{k=1}^{K} \mathbb{E}[n_k] \Delta_k
\]

Rewriting the regret in this way suggests that the goal of any algorithm is to 
quickly learn the arms with large \( \Delta_k \) and discard them. Indeed, for arms whose \( \Delta_k \) 
is large, pulling them even a few times causes a high regret.

In general, to study the performance of any given algorithm a theoretical analysis 
is conducted in order to provide an upper and/or lower bound on the magnitude of 
the regret.

2.1.2 A fundamental result

Before introducing any algorithm, it is worth noticing that no matter what strategy 
we pick, there exists a fundamental result which says that the regret of any bandit 
algorithm has to grow at least logarithmically in the number of plays.
We first introduce the notion of Kullback–Leibler divergence [Kullback and Leibler, 1951], which measures the "distance" between two probability distributions. For \( p, q \in [0, 1] \) let:

\[
KL(p, q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}
\]

This value is in the interval \([0, 1]\), with a value of 0 indicating the two distributions being similar, if not identical, while a value of 1 denotes the two distributions being completely different.

We can now establish the fundamental result first described in [Lai and Robbins, 1985] regarding Bernoulli distributions. This distribution is described by a single parameter, the probability of success, which is also the expected value. Then, an instance of the Bernoulli multi-armed bandit problem can be completely described by \( \Theta = (\mu_1, \ldots, \mu_N) \), where \( \mu_i \) is the expected reward of arm \( i \). For a given instance \( \Theta \), the expected regret of an online algorithm can be denoted as \( \mathbb{E}[R(T, \Theta)] \).

**Theorem 2.1.2** ([Lai and Robbins, 1985]). Consider a strategy s.t. \( \forall a > 0 \) and fixed \( \Theta \), we have \( \mathbb{E}[R(T, \Theta)] = o(T^a) \). Then for any instance \( \Theta \) such that \( \mu_i \) are not all equal:

\[
\liminf_{T \to +\infty} \frac{\mathbb{E}[R(T, \Theta)]}{\log(T)} \geq \sum_i \frac{\Delta_i}{KL(\mu^*, \mu_i)}
\]

From this theorem, it can be shown that the regret has a lower bound of \( \sum_i \log(T)/\Delta_i \). Informally, it states that any consistent algorithm must make at least a logarithmic number of mistakes on every instance. Thus, an online algorithm reaching a logarithmic regret is optimal. This lower bound holds more generally than only Bernoulli distributions, as described for example in [Burnetas and Katehakis, 1996].

However, if the gap between arms is very small the regret will not grow infinitely. If we fix a time horizon \( T \), then a lower bound which is distribution-independent can be established as follows:

**Theorem 2.1.3** ([Auer et al., 2002b]). For any number of actions \( K \geq 2 \), there exists a distribution of losses (rewards) such that:

\[
\mathbb{E}[R_T] \geq \frac{1}{20} \min\{\sqrt{KT}, T\}
\]

Thus the regret will scale with the number of actions \( K \).

### 2.2 Algorithms

Next, we are going to introduce the most common algorithms for the stochastic multi-armed bandit problem.

To design efficient algorithms, a natural idea is to consider the sample mean as a proxy for the real (expected) rewards. For every arm \( k \) and time step \( t \), we define the sample mean \( \hat{\mu}_k \) as:

\[
\hat{\mu}_{k,t} = \frac{\sum_{t=1}^{T} r_t 1\{a_t = k\}}{n_{t,k}}
\]
where \( n_{t,k} \) is the number of times arm \( k \) has been played up to time \( t \). To estimate the deviation of the sample mean from the real mean, we can use the Chernoff-Hoeffding ([Hoeffding, 1963]) inequality, assuming the rewards are bounded, i.e. \( r_t \in [0, 1] \).

**Theorem 2.2.1** ([Hoeffding, 1963]). Let \( X_1, \ldots, X_n \) be iid random variables in \([0, 1]\) such that for all \( i \), \( \mathbb{E}[X_i] = \mu \) and let \( \hat{\mu} = (\sum_{i=1}^{n} X_i)/n \) be their sample mean. Then:

\[
\Pr(|\hat{\mu} - \mu| \geq \delta) \leq 2e^{-2n\delta^2}
\]

If we apply this inequality to the multi-armed setting and choose \( \delta = \sqrt{\ln t/n_{k,t}} \) then at any time step \( t \) we will have:

\[
|\hat{\mu}_{k,t} - \mu_k| < \sqrt{\frac{\ln t}{n_{t,k}}}
\]

with probability at least \( 1 - 2/t^2 \). This bound is the main reason behind the UCB1 algorithm (2) we will describe below.

### 2.2.1 \( \epsilon \)-greedy

The simplest algorithm one can think of is one that pulls the best arm after some initial exploration. However, this algorithm suffers a linear regret since, at any time, there is a constant probability of not choosing the best arm. An optimal algorithm should indeed never stop exploring. Thus, another naive alternative is the algorithm described in (1), which at every time step explores an arm randomly selected with some probability \( \epsilon \).

**Algorithm 1 \( \epsilon \)-greedy algorithm**

Input: \( \epsilon, 0 < \epsilon < 1 \)
Initialize: Play each arm once
for \( t = 1, 2, \ldots \) do
  with probability \( 1 - \epsilon \) pull the arm with highest estimate \( \hat{\mu} \)
  with probability \( \epsilon \) pull arm selected uniformly at random
end for

Unfortunately, also this algorithm incurs linear regret. To see this, it suffices to consider that each arm is pulled on average \( \epsilon T/K \) times and the regret is then at least:

\[
\frac{\epsilon T}{K} \sum_{i: \mu^* > \mu_i} \Delta_i
\]

which is linear in \( T \). Although it is not theoretically optimal, this algorithm often does well in practice and is widely used because of its ease of implementation.

### 2.2.2 Upper Confidence Bound

The Upper Confidence Bound algorithm (2) from [Auer et al., 2002a] is based on the principle of *optimism in the face of uncertainty*. It is indeed based on
Algorithm 2 UCB1 algorithm

**Initialize:** Play each arm once

for $t = 1, 2, \ldots$

play arm $j$ that maximizes \( \hat{x}_j + \sqrt{\frac{\ln n}{n_j}} \) where \( \hat{x}_j \) is the average reward obtained from arm $j$, $n_j$ is the number of times arm $j$ has been played so far, and $n$ is the overall number of plays done so far.

end for

choosing the arm which promises the highest reward under optimistic assumptions, e.g. the reward of each arm is as large as possible based on the data that has been observed. The key part of this algorithm is the term \( \sqrt{\ln n/n_j} \), which is a high confidence upper bound of the error on the empirical average reward \( \hat{x}_j \). Intuitively, this term prevents us to play always the same arm without checking the other ones, since when we play one arm $n_j$ increases and the second term decreases.

Then, each round the best possible arm will be chosen based on 2 scenarios: either \( \hat{x}_j \) is large, implying a high reward, or \( \sqrt{\ln n/n_j} \) is large, i.e., $n_j$ is small, implying an under-explored arm. In both cases, this arm is worth choosing. The terms \( \hat{x}_j \) and \( \sqrt{\ln n/n_j} \) represent respectively the amount of exploitation and exploration and summing them up is a natural way to implement the exploration-exploitation tradeoff.

In general, it can be proven that the UCB1 algorithm has a fixed regret.

**Theorem 2.2.2** ([Auer et al., 2002a]). For all $K > 1$, if policy UCB1 is run on $K$ arms having arbitrary reward distributions $P_1, \ldots, P_K$ with support in $[0, 1]$, then its expected regret after any number $n$ of plays is at most

\[
8 \sum_{i: \mu_i < \mu^*} \left( \frac{\ln n}{\Delta_i} \right) + \left( 1 + \frac{\pi^2}{3} \right) \left( \sum_{j=1}^{K} \Delta_j \right)
\]

### 2.2.3 Thompson Sampling

In one of the earliest works on bandits, Thompson [Thompson, 1933] proposed a randomized Bayesian algorithm to minimize the regret. The key idea is to assume a prior distribution on the reward function of every arm and at each time step play an arm according to its posterior probability of being the best arm. This algorithm became known as the Thompson sampling algorithm, and it is part of the so called probability matching algorithms.

In recent years, this algorithm gained a lot of attention due to its success in practical applications as described for example in [Chapelle and Li, 2011], [Scott, 2010]. Some researchers (see [Chapelle and Li, 2011]) claim that the reason for its lack of popularity in the literature was the absence of a strong theoretical analysis. However in the last years different studies faced an accurate analysis of the regret bounds of this algorithm, both from a Bayesian (see [D. Russo, 2016]) and frequentist (see [Agrawal and Goyal, 2013b]) point of view. The latter is a stronger point of view, since the analysis is prior free and makes it directly comparable to other algorithms like UCB.

In (3) the algorithm using Beta priors is illustrated. For this algorithm, an analysis has been conducted in [Agrawal and Goyal, 2013a] showing that the Thompson
sampling algorithm reaches the asymptotic lower bound of [Lai and Robbins, 1985].

Algorithm 3 Thompson Sampling using Beta priors

**Initialize:** For each arm \( i = 1, \ldots, N \) set \( S_i = 0, F_i = 0 \).

for \( t = 1, 2, \ldots \)

for each arm \( i = 1, \ldots, N \), sample \( \theta_i(t) \) from Beta\( (S_i + 1, F_i + 1) \). 
Play arm \( i(t) := \arg \max_i \theta_i(t) \) and observe reward \( r_t \).

if \( r_t = 1 \) then
\( S_{i(t)} = S_{i(t)} + 1 \)
else
\( F_{i(t)} = F_{i(t)} + 1 \).
end if

end for

Theorem 2.2.3 ([Agrawal and Goyal, 2013a]). For the \( N \)-armed stochastic bandit problem, TS using Beta priors has expected regret

\[
\mathbb{E}[R(T)] \leq (1 + \epsilon) \sum_{i=2}^{N} \frac{\ln(T)}{KL(\mu_i, \mu_1)} \Delta_i + O\left(\frac{N}{\epsilon^2}\right)
\]

in time \( T \). The big-Oh notation assumes \( \mu_i, \Delta_i, i = 1, \ldots, N \) to be constants.

2.3 Contextual bandits

In most bandit problems there is information associated with each arm in the beginning of the rounds, which can help making better decisions. For example, in a web article recommender system this information may be related to the current user, his age or location, the time of the day and so on. Knowing this contextual information can certainly help in the choice of the article to be put on the ”frontpage”. In this case, the bandits are said contextual (or with side information) [Li et al., 2010].

Formally, in the \( K \)-action stochastic contextual bandit problem the learner observes a context \( x_t \) from the set of all possible contexts \( C \). Next, it chooses an action \( a_t \in [K] \). To avoid confusion, we will denote the reward by \( r_{t,a} \) instead of \( X_{t,a} \) from now on. We can make the assumption that the reward \( r_{t,a} \) which is incurred satisfies the following:

\[ r_{t,a} = f(x_t, a_t) + \eta_t \]

where \( f \) is the reward function unknown to the learner and \( \eta_t \) is random noise with zero mean, hence \( \mathbb{E}[r_t] = f(x_t, a_t) \). If \( f \) was known to the learner, the optimal choice would be \( a^*_t = \arg \max_{a \in [K]} f(x_t, a) \). The regret due to loss of knowledge can then be written as:

\[ R_T = \mathbb{E} \left[ \sum_{t=1}^{T} \max_{a \in [K]} f(x_t, a) - \sum_{t=1}^{T} r_{t,a} \right] \]
2.4 Stochastic linear bandits

Another important assumption we can make at this point is that the reward function $f$ satisfies a particular linear parametrization:

$$f(x, a) = \theta^T \psi(x, a)$$

where $\theta^T \in \mathbb{R}^d$ is an unknown parameter vector to be learned and $\psi: C \times [K] \to \mathbb{R}^d$ is a feature map. For example, if the context denotes the visitor of a website selling wines, the actions are wines to recommend and the reward is the revenue on a wine sold, then the features could indicate the interests of the visitors as well as the origin of the wine.

In this scenario the identity of the actions becomes less important and we rather let the algorithm choose the feature vectors. Formally, in round $t$ the learner can choose an action $x_t$ from a decision set $C_t \subset \mathbb{R}^d$, incurring the reward:

$$r_{t,a} = \langle x_t, \theta \rangle + \eta_t$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product. The regret can be written as:

$$R_T = \mathbb{E} \left[ \sum_{t=1}^{T} \max_{x \in C_t} \langle x, \theta \rangle - \sum_{t=1}^{T} r_{t,a} \right]$$

Notice in particular that if $C_t = \{e_1, e_2, \ldots, e_d\}$, the original non-contextual framework is recovered.

Since in this case we have a fixed vector of weights to be learned through linear combinations of features, the regret scales with the dimension $d$ and not with the number of actions $K$.

2.4.1 LinUCB / OFUL

The Upper Confidence Bound scheme can be applied to linear bandits as well and different authors have referred to it with different names: LinUCB [Li et al., 2010], OFUL [Abbasi-Yadkori et al., 2011] (Optimism in the Face of Uncertainty for Linear bandits) and LinREL [Auer, 2002].

One popular idea, present within these algorithms, is to use a ridge regression estimator: suppose at the end of round $t$ a bandit algorithm has chosen context vectors $x_1, \ldots, x_t \in \mathbb{R}^d$ and received rewards $r_1, \ldots, r_t$. Then the ridge regression estimate of $\theta^*$ is defined as the minimizer of the penalized squared empirical loss:

$$L_t(\theta) = \sum_{s=1}^{t} (r_s - \langle x_s, \theta \rangle)^2 + \lambda \|\theta\|_2^2$$

where $\lambda \geq 0$ is a penalty factor and $\|\cdot\|_2$ is the Euclidean norm of a vector. By solving $\hat{\theta} = \arg\min_{\theta \in \mathbb{R}^d} L_t(\theta)$, it can be shown that $\hat{\theta}$ satisfies the following:

$$\hat{\theta} = A_t(\lambda)^{-1} \sum_{s=1}^{t} r_s x_s$$

where:
\[ A_t(\lambda) = \lambda I \sum_{s=1}^{t} x_s x_s^T \]

The goal now is to build confidence intervals which contain with high probability the true parameter vector \( \theta^* \). How to get confidence sets? One popular choice is to consider ellipsoid confidence regions ([Abbasi-Yadkori et al., 2011]):

\[ C_t = \{ \theta \in \mathbb{R}^d : \| \theta - \hat{\theta} \|_A^2 \leq \beta \} \]

where \( \| x \|_A = \sqrt{x^T A x} \). It can be shown that with this choice the UCB values assume a simple form:

\[ \text{UCB}_t(x) = \langle a, \hat{\theta} \rangle + \beta^{1/2} \| x \|_{A^{-1}} \]

From this an algorithm can be derived, which is sketched in (4).

**Algorithm 4 LinUCB**

**Input:** \( \alpha > 0, \lambda > 0 \)
**Initialize:** \( A \leftarrow \lambda I_d, b \leftarrow 0_d \)
for \( t = 1, 2, \ldots \)
\( \theta_t \leftarrow A^{-1}b \)
Observe features of all \( K \) arms \( a \in \mathcal{A}_t : x_{t,a} \in \mathbb{R}^d \)
for each arm \( i = 1, \ldots, K \)
\( s_{t,a} = x_{t,a}^T \theta_t + \alpha \sqrt{x_{t,a}^T A_t^{-1} x_{t,a}} \)
end for
Pull arm \( a_t = \arg \max_a s_{t,a} \), break ties arbitrarily.
Receive reward \( r_t \in [0, 1] \).
\( A \leftarrow A + x_{t,a} x_{t,a}^T \)
\( b \leftarrow b + x_{t,a} r_t \)
end for

**Theorem 2.4.1** ([Chu et al., 2011]). Suppose the rewards \( r_{t,a} \) are independent random variables with mean \( \mathbb{E}[r_{t,a}] \). Then, with probability \( 1 - \delta/T \), we have for all \( a \in [K] \) that:

\[ |\hat{\theta}_{t,a} - \langle x, \theta_a \rangle| \leq (\alpha + 1) s_{t,a} \]

where \( s_{t,a} = \langle x_{t,a}, \theta_t \rangle + \alpha \sqrt{x_{t,a}^T A_t^{-1} x_{t,a}} \). Furthermore, if \( \alpha = \sqrt{\frac{1}{2} \ln \frac{2TK}{\delta}} \), then with probability at least \( 1 - \delta \), the regret of the algorithm is:

\[ O(\sqrt{Td \ln^3 (KT \ln(T)/\delta)}) \]

However, the rewards in the above theorem are not independent random variables, since LinUCB algorithm uses samples from previous rounds to estimate \( \theta_a \). In [Abbasi-Yadkori et al., 2011] a new martingale based technique is used to show that similar results can be obtained without the assumption of independent random variables.
2.4.2 Linear Thompson Sampling

On the other hand, contextual bandits can also be treated from a Bayesian point of view. The general framework for Thompson sampling with contextual bandits ([Agrawal and Goyal, 2013b]) assumes a parametric likelihood function for the reward $P(r|a, x, \theta)$ where $\theta$ is the model parameter. If the true parameter $\theta_*$ was known, we would choose the arm which maximizes the expected reward $\max_a \mathbb{E}[r|a, x, \theta_*]$. Instead, with Thompson sampling we have a prior $P(\theta)$ on the model parameter $\theta$ and based on the data observed, we update its posterior distribution by $P(\theta|D) \propto P(\theta) \prod_{t=1}^{T} P(r_t|x_t, a_t, \theta)$.

Following the approach in [Agrawal and Goyal, 2013b], we can assume the reward for arm $i$ at time $t$ is generated from an unknown distribution with mean $x_{t,a}^T \theta$, where $\theta \in \mathbb{R}^d$ is a fixed but unknown parameter. The noise $\eta_{t,a} = r_{t,a} - x_{t,a}^T \theta_*$ is assumed to be conditionally $R$-sub-Gaussian for a constant $R \geq 0$, i.e.:

\[ \forall \lambda \in \mathbb{R}, \mathbb{E}[e^{\lambda \eta_{t,a} \mid \{x_{t,a}\}_{s=1}^{k}, \mathcal{H}_{t-1}}] \leq \exp\left(\frac{\lambda^2 R^2}{2}\right) \]

Based on the different prior and likelihood functions which satisfy the sub-Gaussian assumption, we can have different versions of the Thompson sampling algorithm. In [Agrawal and Goyal, 2013b] Gaussian prior and likelihood functions are used to make the analysis simpler, but it is stressed how the analysis is unrelated to the actual reward distribution. Similarly to the UCB algorithms for linear bandits using ridge regression, we have the following definitions:

\[ A_t = I_d + \sum_{s=1}^{t} x_{s,a} x_{s,a}^T \]

\[ \hat{\theta}_t = A_t^{-1} \sum_{s=1}^{t} x_{s,a} r_{s,a} \]

We can assume a Gaussian likelihood function for the reward, e.g. $r_{t,a} \sim \mathcal{N}(\theta^T x_{t,a}, v^2)$ with $v^2 = R \sqrt{\frac{2d}{\epsilon} D \ln \frac{1}{\delta}}$ as an input parameter for the algorithm. Then, if the prior for $\theta$ at time $t$ is given by $\mathcal{N}(\hat{\theta}_t, v^2 A^{-1}_t)$, it can be shown that the posterior distribution at time $t + 1$ will be $\mathcal{N}(\hat{\theta}_{t+1}, v^2 A^{-1}_{t+1})$.

In the actual algorithm, we can simply generate a sample $\tilde{\theta}_t$ from $\mathcal{N}(\hat{\theta}_t, v^2 A^{-1}_t)$ and play the arm $i$ which maximizes $x_{t,a} \tilde{\theta}_t$. An outline of the algorithm is given in (5).

**Theorem 2.4.2** ([Agrawal and Goyal, 2013b]). For the stochastic contextual bandit problem with linear payoff functions, with probability $1 - \delta$, the total regret in time $T$ for Thompson Sampling is bounded by $O\left(\frac{d^2}{\epsilon} \sqrt{T^{1+\epsilon}} ln(T) \ln \frac{1}{\delta}\right)$, for any $0 < \epsilon < 1, 0 < \delta < 1$. Here, $\epsilon$ is a parameter used by the Thompson Sampling algorithm.

In the proof of the regret bound in [Agrawal and Goyal, 2013b], arms are divided into saturated and unsaturated, based on whether the standard deviation of the estimates of the mean for an arm is smaller or larger compared to the standard deviation for the optimal arm. Then they show how to give bounds on the probability of playing arms belonging to each of those groups.

There are various reasons for the recent success of Thompson sampling algorithms. For example, when considering the running time, these algorithms often
Algorithm 5 Thompson Sampling with linear payoffs

Input: $\delta \in [0, 1]$, $\epsilon \in [0, 1]$

Initialize: $v = \sqrt{\frac{24}{\epsilon} d \ln \frac{t}{\delta}}$, $A = I_d$, $\hat{\theta} = 0_d$, $b = 0_d$

for $t = 1, 2, ...$

\[ \text{sample } \tilde{\mu} \text{ from distribution } \mathcal{N}(\hat{\mu}, v^2 A^{-1}) \]

Pull arm $a_t = \arg\max_a x_{t,a}^\top \hat{\theta}$

Receive reward $r_t$

\[ b \leftarrow b + x_{t,a_t} r_t \]

Update $A \leftarrow A + x_{t,a_t} x_{t,a_t}^\top$

Update $\hat{\theta} \leftarrow A^{-1} b$

end for

offer more scalable solutions compared to UCB ones, as shown in [Kocak et al., 2014] and [S. Vaswani, 2017], mainly since they do not have to calculate any confidence bound. Also, from a practical point of view randomized algorithms seem to perform better (see for example [Chapelle and Li, 2011]) when a delay is present. This is often the case in a real world system, where the feedback arrives in batches over a certain period of time. In this case the advantage of algorithms like Thompson sampling is that randomizing over actions alleviates the influence of delayed feedback. On the other hand, deterministic algorithms based on UCBs suffer a larger regret in case of sub-optimal choices.

2.5 Bandits on graphs

An active line of research in bandit algorithms for recommender systems is focused on models where users and/or items with their interactions can be represented by a graph. In this setting, the notion of smoothness plays an important role: a smooth graph function is a function on a graph that returns similar values on neighboring nodes. For example, interests of people in a social network tend to change smoothly, since friends are more likely to have similar preferences.

Ignoring any kind of graph structure and other information and using known algorithms could lead to a very large regret if the number of actions $K$ is very large. For example, [Valko et al., 2014] consider a scenario like movie recommendations, where the number of actions $K$ (e.g. movies) is much greater than the time horizon $T$ and the learner does not have the budget (in this case time) to try all the options. In this setting, the arms are the nodes of a graph and the expected payoff of pulling an arm is a smooth function on this graph. They introduce a new quantity called effective dimension $d$, which is smaller than $K$ when $K \gg T$, and give a UCB-algorithm which exploits the spectral properties of the graph and matches a lower bound of $\Omega(\sqrt{dT})$. On the other hand, its Bayesian version ([Kocak et al., 2014]) incurs a slightly worse theoretical regret but it is computationally less expensive, since each round it does not have to calculate a confidence bound for every arm.

A different scenario is adopted in [Cesa-Bianchi et al., 2013], where users are nodes on a graph and a bandit instance is allocated on each node, allowing information sharing between different nodes. In particular, in the gang of bandits model ([Cesa-Bianchi et al., 2013]) each node is described by a parameter vector $w_i$, which
is assumed to be smooth given the Laplacian of the graph. Indeed, at each iteration of the algorithm after the current user is served using a UCB policy, a local update of $w_i$ is performed involving nearby users’ $w_j$. The main drawback of this approach is that it depends quadratically on the number of nodes. A recent work in the same setting from [S. Vaswani, 2017] using ideas from Gaussian Markov Random fields and Thompson sampling shows that the computational workload can be reduced, resulting in a more scalable algorithm which can also be used with very large graphs.

A stronger assumption is contained in other works such as [Gentile et al., 2014] or [Li et al., 2016], where nodes (which represent users or items, or both of them) are clustered in different groups and nodes in the same cluster are supposed to exhibit the same behaviour. In particular, the regret of the CLUB algorithm from [Gentile et al., 2014] scales roughly with the number of clusters.

### 2.5.1 Context-Dependent Clustering of Bandits

The assumption of clustering nodes is also contained in what is the closest work to this thesis project, the algorithm CAB described in [Gentile et al., 2017].

The setting is the following: each user is described by an unknown parameter vector determining their behaviour, denoted as $u_i$. Also, a $\gamma$-gap assumption is present: users are clustered in groups according to a certain $\gamma$ parameter given as input to the algorithm. The algorithm receives a user with a set of contexts at each iteration and builds different user’s neighbourhoods depending on the context, using a UCB-scheme: the idea is that different contexts induce different clusters across users. After the user is served using a collaborative approach, not only him but also its neighbourhood is updated.

This approach shows good empirical results (see [Gentile et al., 2017]), but scales poorly when the number of users increases, forcing to use other techniques which may reduce accuracy, such as sampling the number of users to build the clusters. This algorithm is described in (6).

In the theoretical analysis of this algorithm, a new quantity is introduced which measures the hardness of the data at hand. For an observed sequence of users $\{i_t\}_{t=1}^T = \{i_1, \ldots, i_T\}$ and corresponding sequence of item sets $\{C_t\}_{t=1}^T = \{C_1, \ldots, C_T\}$, where $C_t = \{x_{t,1}, \ldots, x_{t,c_t}\}$, the hardness $\text{HD}(\{i_t, C_t\}_{t=1}^T, \eta)$ of the pairing $\{i_t, C_t\}_{t=1}^T$ at level $\eta > 0$ is defined as:

$$\text{HD}(\{i_t, C_t\}_{t=1}^T, \eta) = \max\{t = 1, \ldots, T : \exists j \in U, \exists k_1, k_2, \ldots, k_t \text{ : } I + \sum_{s \leq t, i_s = j} x_{s,k_s} x_{s,k_s}^\top \text{ has smallest eigenvalue } < \eta\}$$

This quantity roughly measures the amount of time (i.e. rounds) we need to wait in the worst case until all the matrices $M_j$ have eigenvalues lower bounded by $\eta$.

Based on this definition, the following theorem is given.

**Theorem 2.5.1** ([Gentile et al., 2017]). Let CAB be run on $\{i_t, C_t\}_{t=1}^T$, with $c_t \leq c$ for all $t$. Also, let the condition $|u_j^\top x - w_j^\top x| \leq CB_j(x)$ hold for all $j \in U$ and $x \in \mathbb{R}^d$, along with the $\gamma$-gap assumption. Then the cumulative regret of the algorithm can be deterministically upper bounded as follows:
Algorithm 6 Context-Aware clustering of Bandits (CAB)

**Input:** Separation parameter $\gamma$, exploration parameter $\nu$

**Init:** $b_i = 0_d$, $M_i = I_d$, $w = 0_d$, $i = 1, ..., n$

for $t = 1, 2, ...$ do
  Set $w_i = M_i^{-1}b_i$ for all users
  Receive user $i_t \in \mathcal{U}$, and context vectors $C_t = (x_1, ..., x_K)$
  Use $CB_i(x) = \alpha(t)\sqrt{x^\top M_i^{-1}x}$ for all $x, i = 1, \ldots, n$
  for $k = 1, 2, ..., K$ do
    Compute neighborhood $\hat{N}_k := \hat{N}_{i_t}(x_k)$ for this item
    $\hat{N}_k = \left\{ j \in \mathcal{U} : |w_i^\top x_k - w_j^\top x_k| \leq CB_i(x_k) + CB_j(x_k) \right\}$
    Set $w_{\hat{N}_k} \leftarrow \frac{1}{|\hat{N}_k|} \sum_{j \in \hat{N}_k} w_j$
    Set $CB_{\hat{N}_k}(x_k) \leftarrow \frac{1}{|\hat{N}_k|} \sum_{j \in \hat{N}_k} CB_j(x_k)$
  end for
  Recommend item $i_{k_t} \in C_t$ such that:
  
  $$k_t = \arg\max_{k=1,\ldots,K} (w_{\hat{N}_k}^\top x_k + CB_{\hat{N}_k}(x_k))$$

  Observe payoff $y_t$

  if $CB_{i_t}(\pi_{k_t}) \geq \gamma/4$ then
    Update $M_{i_t} \leftarrow M_{i_t} + \pi_{k_t}\pi_{k_t}^\top$
    Update $b_{i_t} \leftarrow b_{i_t} + y_t\pi_{k_t}$
  else
    for $j \in \hat{N}_{k_t}$ such that $CB_j(\pi_{k_t}) < \gamma/4$ do
      Set $M_{j} \leftarrow M_{j} + \pi_{k_t}\pi_{k_t}^\top$
      Set $b_{j} \leftarrow b_{j} + y_t\pi_{k_t}$
    end for
  end if
end for
The regret is then composed of two terms: the first measures the hardness of the data according to the definition given above, while the second is a term depending on $\sqrt{T}$ which can be found also in other works regarding linear bandits, such as [Abbasi-Yadkori et al., 2011] or [Chu et al., 2011]. On the other hand, in the second term the number of users $n$ to be served is replaced by a smaller quantity, the ratio $\frac{n}{|N_{i_t}(\bar{x}_{k_t})|}$, which depends on the size of clusters built.
Chapter 3

Method

In this chapter we describe the algorithm that we developed with its modifications and how we practically implemented it. In particular, we remind that our aim is to provide a Bayesian counterpart of the algorithm proposed in [Gentile et al., 2017]. Thus, our algorithm strictly follows the setting there depicted (see subsection 2.5.1). Furthermore, we describe the methodology that we used to evaluate our algorithm and to compare it to other known algorithms. In particular, we describe how we prepared the datasets used and what metric is adopted to evaluate the results.

3.1 ThompCAB

We call our algorithm ThompCAB, which stands for *Thompson Sampling Context Aware clustering of Bandits*. We are going to describe it below.

3.1.1 Setting

Every user is described by an unknown parameter vector $\theta_i$ which determines their behaviour. As typically happens in the online learning scenario, the learning process sequentially happens in rounds. The algorithm at each iteration receives a user index $i_t$ from a set of users $\mathcal{U}$, together with a set of context vectors $C_t = \{x_1, \ldots, x_k\}$ describing the different items we can recommend to the current user.

The algorithm then selects one of these arms and receives a stochastic reward, expected value of which is an unknown linear function of the action, i.e. $r_{t,a} = \theta_i^\top x_a$. The goal of the algorithm is to minimize the regret (see section 2.4) over $T$ time steps:

$$R_T = \mathbb{E}\left[ \sum_{t=1}^{T} \max_{x \in C_t} \theta_i^\top x - \sum_{t=1}^{T} r_{t,a} \right]$$

3.1.2 Algorithm description

Following the approach of Gentile et al. [2017], the main goal of the algorithm is to build some ”neighborhoods” for the user served at time step $t$, based on the items it has at disposal. The idea is that different users agree on their opinion on certain items and disagree on others. If two users in the neighborhoods are similar up to a certain threshold, then the parameters of both of them will be updated. The
assumption is that users lying within the same cluster will have similar behaviours. Formally, if two users $i,j$ are in the same cluster with respect to an item $x$, then $\tilde{\theta}_i^\top x = \tilde{\theta}_j^\top x$. If this is not verified, then there exists a gap parameter $\gamma$ such that $|\tilde{\theta}_i^\top x - \tilde{\theta}_j^\top x| \geq \gamma$. This $\gamma$-gap assumption is also present in similar works such as [Gentile et al., 2014], [Li et al., 2016] and [Gentile et al., 2017].

Since we want to operate in a Bayesian environment, following the approach of [Agrawal and Goyal, 2013b] for each user we adopt a Gaussian likelihood function for the unknown reward function at time $t$: given a context $x_t$ and unknown parameter $\theta$, then $r_{t,a} \sim \mathcal{N}(x_t^\top \theta, v^2)$, where $v$ is a fixed parameter of the algorithm. Then, we define the following:

$$B_t = I_d + \sum_{s=1}^t x_{s,a}\! x_{s,a}^\top$$

$$\tilde{\theta}_t = B_t^{-1} \sum_{s=1}^t x_{s,a} r_{s,a}$$

If the prior for $\theta$ at time $t$ is given by $\mathcal{N}(\tilde{\theta}_t, v^2 B_t^{-1})$, we can compute the posterior distribution at time $t + 1$:

$$P(\tilde{\theta}|r_{t,a}) \propto P(r_{t,a}|\theta) P(\theta)$$

$\propto \exp \left\{ -\frac{1}{2v^2} (r_{t,a} - \tilde{\theta}_{t,a})^2 + (\theta - \tilde{\theta}_t)^\top B_t (\theta - \tilde{\theta}_t) \right\}$

$\propto \exp \left\{ -\frac{1}{2v^2} (r_{t,a}^2 + \tilde{\theta}_{t,a}^\top x_{t,a} x_{t,a}^\top \tilde{\theta} + \tilde{\theta}^\top B_t \tilde{\theta} - 2\tilde{\theta}^\top x_{t,a} r_{t,a} - 2\tilde{\theta}^\top B_t \tilde{\theta}_t) \right\}$

$\propto \exp \left\{ -\frac{1}{2v^2} (\tilde{\theta}_{t+1}^\top \tilde{\theta} + 2\tilde{\theta}^\top B_{t+1} \tilde{\theta}_{t+1}) \right\}$

$\propto \mathcal{N}(\tilde{\theta}_{t+1}, v^2 B_{t+1}^{-1})$

At every time step we generate a sample $\tilde{\theta}$ for each user from the distribution $\mathcal{N}(\tilde{\theta}_{t,t}, v^2 B_{t,t}^{-1})$, which we then use to build the neighbourhood. To this aim, we use a confidence bound, which we define in the following way:

$$CB_i(x_a) = |\tilde{\theta}_i^\top x_a - \tilde{\theta}_i^\top x_a|$$

Intuitively, this difference is an expression of the uncertainty we have about our belief on the score of a certain item. If indeed the variance of the distribution we use for sampling $\tilde{\theta}$ (parametrized by $v^2 B^{-1}$) is large, then the $CB$ term will more likely be large, meaning we are more uncertain about this choice. There could be other ways to calculate confidence bounds in a Bayesian setting, for example as done in [Kaufmann et al., 2012]. However, we adopted this approach mainly for its ease of computation.

Then, in order to build the neighborhoods we compare the estimated rewards for the current user and any other user to their confidence bounds:

$$|\theta_i^\top x - \theta_j^\top x| \leq CB_i(x) + CB_j(x)$$
If this condition is verified, then the user $j$ is included in the neighbourhood computed with respect to item $x$.

Once we have the neighbourhoods, to get the score for a particular item $x_{t,a}$, we define a neighbourhood parameter $\tilde{\theta}_{N_K}$ as the mean of parameters $\tilde{\theta}_j$ of all users $j$ in the neighbourhood:

$$\tilde{\theta}_{N_K} = \frac{1}{|N_K|} \sum_{j \in N_K} \tilde{\theta}_j$$

Then, in the algorithm we can simply generate a sample $\tilde{\mu}_t$ from $\mathcal{N}(\tilde{\theta}_t, v^2 B_t^{-1})$ and play the arm $i$ which maximizes $x_{t,a}^T \tilde{\theta}_t$.

Once we have $\tilde{\theta}_{N_i}$ for all the neighbourhoods $i = 1, \ldots, K$ we can calculate the score for all the items and recommend the item with the highest score:

$$x_{k_t} = \arg \max_{k=1,\ldots,K} \tilde{\theta}_{N_K}^T x_k$$

After we get the feedback $r_{k_t}$ from the environment, we can update the parameters by solving the ridge regression problem involving items served previously, as described in subsection 2.4.1. In particular, if the user is “confident” enough about the recommended item, e.g. $CB_i(x_{k_t}) < \gamma$, we update not only him but also the users in his neighborhood which satisfy $CB_j(x_{k_t}) < \gamma$. The algorithm is sketched in (7).

### 3.1.3 Keeping count of the number of updates

A second variant of the algorithm takes into account the number of times a user has been updated in the update subroutine. Indeed, it can happen that we have some users which we know well (i.e. we served them many times) in a neighbourhood and we do not want to update them based on a feedback from a new user, which we know less well. For this reason, we modify the threshold used for the update by using $CB_j(x_{k_t}) < \gamma / n_j$, where $n_j$ is the number of times user $j$ has been served. This modified version of the update subroutine is sketched in 8.

### 3.1.4 Implementation

For the implementation, we first allocate one bandit instance per user, assuming the number of users $N$ is known in advance. We made this assumption in order to get an easier implementation but it can be easily relaxed and users can be added on the fly. In particular, the parameters $\tilde{\theta}$ and $\tilde{\theta}$ are stored in $N \times d$ matrices while the inverse matrices $B^{-1}$ are stored in a tensor $N \times d \times d$. Then, at every iteration the algorithm receives a user to serve, together with a set of contexts. To store the neighborhoods we use a $N \times K$ matrix initialized to all 0, where $K$ is the number of actions. Then, each column represents the neighborhood for a given context. If user $n$ is in the neighborhood for the action $k$, then $N[n,k]$ will be set to 1. Clearly, the row corresponding to the current user contains only 1’s. In this way, all the computations are reduced to matrix multiplications.
Algorithm 7 Thompson CAB

**Input:** Separation parameter $\gamma$, exploration parameter $v$

**Init:** $b_i = 0_d$, $M_i = I_d$, $\tilde{\theta} = 0_d$, $i = 1, ..., n$

**for** $t = 1, 2, ...$ **do**

- Sample $\tilde{\theta}_i$ from $\mathcal{N}(\tilde{\theta}_i, v^2 M_i^{-1})$ for all users
- Receive user $i_t \in \mathcal{U}$, and context vectors $C_t = (x_1, ..., x_K)$

**for** $k = 1, 2, ..., K$ **do**

- Compute neighborhood $\hat{N}_k := \hat{N}_i(x_k)$ for this item
  \[
  \hat{N}_k = \left\{ j \in \mathcal{U} : |\tilde{\theta}^{\top} x_k - \hat{\theta}^{\top} x_k| \leq CB_i(x_k) + CB_j(x_k) \right\}
  \]
- Set $\hat{\theta}_{\hat{N}_k} \leftarrow \frac{1}{|\hat{N}_k|} \sum_{j \in \hat{N}_k} \tilde{\theta}_j$

**end for**

**end for**

Recommend item $\bar{x}_{k_t} \in C_t$ such that:

\[
  k_t = \arg\max_{k=1,...,K} \hat{\theta}^{\top}_{\hat{N}_k} x_k
\]

Observe reward $r_{k_t}$

Update $M_{i_t} \leftarrow M_{i_t} + \bar{x}_{k_t} \bar{x}_{k_t}^{\top}$

Update $b_{i_t} \leftarrow b_{i_t} + r_{k_t} \bar{x}_{k_t}$

**if** $CB_{i_t}(\bar{x}_{k_t}) \leq \gamma/4$ **then**

**for** $j \in \hat{N}_{k_t}$ such that $CB_j(\bar{x}_{k_t}) < \gamma/4$ **do**

- Set $M_j \leftarrow M_j + \bar{x}_{k_t} \bar{x}_{k_t}^{\top}$
- Set $b_j \leftarrow b_j + r_{k_t} \bar{x}_{k_t}$

**end for**

**end if**

**end for**

Algorithm 8 Alternative version of the update subroutine in Thompson CAB

**Input:** user $i_t$, context $\bar{x}_{k_t}$, reward $r_{k_t}$, neighbourhood $\hat{N}_{k_t}$, $M_i$, $b_i$, $i = 1, ..., n$

Update $M_{i_t} \leftarrow M_{i_t} + \bar{x}_{k_t} \bar{x}_{k_t}^{\top}$

Update $b_{i_t} \leftarrow b_{i_t} + r_{k_t} \bar{x}_{k_t}$

**if** $CB_{i_t}(\bar{x}_{k_t}) \leq \gamma/4$ **then**

**for** $j \in \hat{N}_{k_t}$ such that $CB_j(\bar{x}_{k_t}) < \gamma/n_j$ **do**

- Set $M_j \leftarrow M_j + \bar{x}_{k_t} \bar{x}_{k_t}^{\top}$
- Set $b_j \leftarrow b_j + r_{k_t} \bar{x}_{k_t}$

**end for**

**end if**
3.1.5 Efficiency

By profiling the algorithm, it turns out there are two bottlenecks: the first is relative to the sampling process of the parameter vectors $\tilde{\theta}$ in each timestep, while the second regards the updating subroutine for all the users. Regarding the former, a key observation which would speed up the process is that not all the bandit instances are updated at every timestep, so in the next round of the algorithm we can still use the previously computed sample (since prior and likelihood function would be exactly the same). However, if the number of users to update is large, we would still have problems. A possible solution could be to properly tune the parameters so that only few users are included in the neighborhoods. Still, if the overall number of users is large (as it realistically happens in a recommender system) this strategy is destined to fail. A reasonable choice would be to use a fraction of the users representing the whole pool. Again, this can be done in a lot of different ways, but we used a straightforward one: random sampling users according to an acceptance probability $p$, with $0 < p \leq 1$. At every time step we sample from a uniform distribution between 0 and 1: if the sample is greater than $p$ then the user is not considered, otherwise we use it for building the neighbourhoods. Obviously 0 is equivalent to not considering any other user than the one who is currently served, while 1 is equivalent to considering the entire pool of users.

3.2 Algorithms

In this section we list the algorithms that we use for comparisons. They are all part of the domain of stochastic linear bandits:

- **ThompsonOne** - a unique instance of Thompson sampling algorithm (5) which makes no difference between users.
- **ThompMulti** - Thompson sampling (5) individual instances for each user, independent of each other.
- **ThompCAB** - our algorithm, described in section 3.1.
- **CAB** - the UCB version (6) of our algorithm, described in subsection 2.5.1.
- **LinUcbOne** - a unique instance of LinUCB algorithm (4) which makes no difference between users.
- **LinUcbMulti** - LinUCB (4) individual instances for each user, independent of each other.
- **random** - an algorithm which at every iteration randomly selects one of the arms.

3.3 Choice of the parameters

We acknowledge that all the algorithms we use depend heavily on the choice of the hyperparameters. In particular, in UCB-like algorithms the parameter controlling the behaviour of the algorithm is $\alpha$ (see in 4): higher values of $\alpha$ produce higher
confidence bounds resulting in a greater amount of exploration, while lower values push the algorithm towards exploitation. On the other hand, Thompson sampling algorithms as well exhibits the same behaviour through the choice of the parameter $v$ (see in 5), which controls the variance in the Normal distribution used for sampling. Furthermore, the algorithms clustering the users such as CAB and ThompCAB have also the separation parameter $\gamma$ which determines the number of users in the neighbourhood to be updated at every time step. This parameter should be carefully tuned, since setting a high value would lead to a lot of updates, thus making the algorithms significantly slower. On the other hand, a low value would remove the clustering benefits.

As done in [Gentile et al., 2017], to properly tune all the parameters we use the 20% of the original dataset, while the remaining 80% is used to report results. In particular, we use a grid search: given a set of parameters, we try different combinations until we find the best one.

3.4 Datasets

To evaluate our algorithm, we test it both on synthetic and real world data. However, the latter case is more difficult for several reasons. First of all, it is difficult to find a publicly available dataset for contextual bandit evaluation, especially when the identity of users is involved. Most companies are indeed not willing to share their data about users. Secondly, even when data is provided, most of the features, if not all, are anonymized, making it more difficult to carry out an unbiased evaluation and infer conclusions.

To our knowledge, the only public dataset specifically built for unbiased evaluation of contextual bandit algorithms is the one released by Yahoo [Chu et al., 2009] regarding its news front page data. However, in our case this data cannot be used since user ids are specifically anonymized so that individual users cannot be identified. Our choice then relied on Avazu dataset, a public dataset used for a Kaggle competition in 2014.

We are going to illustrate the preprocessing of data and the setup for our experiments in the following sections.

3.5 Artificial Data

In order to generate synthetic data, we adopt the same technique as in [Zhou and Brunskill, 2016]. We artificially create different latent models for users where each model has a fixed number of parameters, and some of them are assigned higher weights. In particular, we use $N = 5$ classes and each latent model $i \in \{1, \ldots, 5\}$ is associated with a weight vector $\beta_i \in \mathbb{R}^d$. The classes and their weights are shown in Figure 3.1.

Users are sampled uniformly at random from these latent models. For each user interaction we generate 20 arms, i.e. $K = 20$. Each arm is associated with a feature vector $x_{ta}$, which is sampled uniformly from $[-1,1]^{10}$ and normalized in order to have $||x_{ta,a}||^2 = 1$. Finally, the reward of each arm is sampled from $N(\beta^T x_{ta,a}, \sigma^2)$ with $\sigma = 0.1$. 24
This is an idealized scenario since in real world we may rarely have users with exactly identical features. Moreover individual features may be subject to variations along time (since user tastes may change).

We study the behaviour of our algorithm in 3 experiments on artificial data:

- Experiment 1: studying the gap parameter $\gamma$ (section 4.1)
- Experiment 2: varying the number of interactions per user (section 4.2)
- Experiment 3: varying the number of users (section 4.3)

### 3.6 Avazu

To test the algorithm on real-world data, the chosen dataset is from a Kaggle competition\(^1\) on Click-Through Rate (CTR) prediction released by Avazu, a digital marketing company specializing in cross-device advertising. The data is collected from 10 days of click-through data, ordered chronologically. The publisher states that non-clicks and clicks are subsampled according to different strategies.

We have different fields and all of them are categorical:

- **id**: ad identifier
- **click**: 0/1 for non-click/click
- **hour**: format is YYMMDDHH, so 14091123 means 23:00 on Sept. 11, 2014 UTC.
- **C1** – anonymized categorical variable

---

\(^1\)https://www.kaggle.com/c/avazu-ctr-prediction
For the preprocessing steps, we mostly follow the approach of the winners of this competition since the code is publicly available on Github\(^2\). First of all, to define the id of the user we combine both device_ip and device_id features. Then, by gathering information from site and app fields we generate some new fields describing the publisher. Finally, the anonymous categories C14-C21 are discarded.

The original dataset contains 40 million rows. After the described steps there are a lot of duplicates which we remove, resulting in 28 million rows left. As an additional step, we decide to remove non-common users, i.e. we only keep users which have at least 20 interactions and at least one interaction has a recorded 1 in the click field. This final dataset contains 4 million rows and 13 columns.

Not surprisingly, most of the users have few interactions with the system, as shown in Figure 3.2.

### 3.6.1 Feature Hashing

A common approach to deal with categorical features is to use one-hot encoding, also known as dummy coding. If a categorical variable can take c values, then c binary features are introduced, the \(i^{th}\) one being the indicator function of the \(i^{th}\) value. For example, if a feature can take 4 values and in one of the records it takes the 3\(^{rd}\) one, then it will be encoded as follows:

\[(0, 0, 1, 0)\]

\(^2\)https://github.com/guestwalk/kaggle-avazu
So, if we have $N$ features and each feature can take $c_n$ values then the dimension of the feature vectors would be:

$$d = \sum_{n=1}^{N} c_n$$

We can quickly analyze our dataset in order to find the number of unique values the features can take. This is reported in Table 3.1.

Using one-hot encoding on these features would lead to feature vectors of dimensionality $d > 8000$, which is clearly a huge dimension for the vectors to be stored and would make the algorithms very slow.

A possible solution, made popular by the software Vowpal Wabbit [Langford et al., 2007] is to use the hashing trick described in [Weinberger et al., 2009]. The idea is to use a hash function to reduce the number of values a feature can take. The final dimension $d$ of the feature vectors represents the number of bins we use for hashing and should obviously be lower than the one we would have with one-hot encoding. We still adopt the one-hot encoding on the hashed features, however in this way we get a compressed representation of the data.

<table>
<thead>
<tr>
<th>Feature</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hour</td>
<td>240</td>
</tr>
<tr>
<td>Banner_pos</td>
<td>7</td>
</tr>
<tr>
<td>device_model</td>
<td>1933</td>
</tr>
<tr>
<td>device_conn_type</td>
<td>4</td>
</tr>
<tr>
<td>C14</td>
<td>2066</td>
</tr>
<tr>
<td>C17</td>
<td>382</td>
</tr>
<tr>
<td>C20</td>
<td>143</td>
</tr>
<tr>
<td>C21</td>
<td>55</td>
</tr>
<tr>
<td>pub_id</td>
<td>3117</td>
</tr>
<tr>
<td>pub_domain</td>
<td>1459</td>
</tr>
<tr>
<td>pub_category</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 3.1: Unique values features can take.
When hashing features, there are two possible ways of doing it:

1. Hash each feature $n$ into a $d_n$-dimensional space and concatenate the outputs. This results in feature vectors of dimension $\sum d_n$.

2. Hash all the features into the same space, using a different hash function for each feature.

We adopt the latter (described in 9), as it is easier to implement. The hash function is the same used in Vowpal Wabbit, called Murmurhash\(^3\). An easy way to get different hash functions for each feature is to concatenate the value $n$ to the feature $v_n$.

**Algorithm 9 Hashing trick**

**Input**: Values for the $N$ features, $v_1, \ldots, v_N$; family of hash functions $h_n$; number of bins $d$.

**Init**: $x_i = 0$, $i = 1, \ldots, n$

**for** $n = 1, \ldots, N$ **do**

\[ i = h_n(v_n) \mod d + 1 \]

\[ x_i = x_i + 1 \]

**end for**

An issue we could encounter is the problem of collisions: multiple hashed features could go in the same bucket. However, as explained in [Chapelle et al., 2015] this is not a major concern.

### 3.6.2 Feature conjunctions

Another element that could improve the performance of algorithms is to consider feature conjunctions. A linear model can indeed learn only independent effects of features but nothing about their combined effect. For example, in the specific case of advertising we could have two fields such as advertiser and publisher. A model without conjunctions would only learn that some advertisers and some publishers have higher Click-Through Rate (CTR) but not for example that the CTR of Netflix could be much higher on a website like bbc.entertainment.

We then combine all the features by 2 by 2 (higher combinations could be also used but we did not try them) and test the effectiveness of this technique. This is equivalent to use a polynomial kernel of degree 2. In this case, feature hashing is even more important to reduce the dimensionality of the data. Indeed, by taking combinations of the 10 initial features in groups of 2 we get:

\[
\binom{10}{2} = \frac{10!}{2!(10-2)!} = \frac{10 \times 9}{2} = 45
\]

Thus we have $10 + 45$ features. To balance between storage and accuracy we need to select a proper dimension $d$ for the hashing process. We thus select $d = 60$. When instead we do not consider feature conjunctions we simply set $d = 30$.

\(^3\)https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp
3.6.3 Policy evaluation

An important discussion often encountered in the field of contextual bandit is related to the evaluation process. It is indeed difficult to effectively evaluate the algorithms without introducing any bias.

To this extent, in recent years a lot of studies have been conducted in order to provide unbiased estimators, such as [Hallak and Mannor, 2017], [Wang et al., 2017] and [Dudik et al., 2011]. Probably the most adopted technique for offline evaluation of policies on real world data is the one based on rejection sampling used in [Li et al., 2010]. They consider the situation when a sequence of events is available, deriving from the interaction with the world of a logging policy, which at each time step chose each arm uniformly at random. Assuming that there is an unknown distribution $D$ from where tuples of contexts and rewards $(x_1, ..., x_k, r_1, ..., r_k)$ are drawn iid. Given a policy $\pi$ and a stream of logged events, then each time step if the policy $\pi$ pulls the same arm as the logging policy the event is retained, otherwise it is discarded and the evaluation process goes to the next record.

This approach is effective, although it significantly reduces the size of the actual data. We follow a similar approach to generate different datasets: we first group the records based on the id of the user, then for every user we split the interactions between clicks ($\text{click} = 1$) and non-clicks ($\text{click} = 0$). For each click we build a round for the algorithms by padding to it 19 other non-clicks sampled uniformly at random from the entire pool of non-clicks. However, differently from [Li et al., 2010] we retain all the records. Furthermore, since we group together ads shown at different hours, we discard this feature pretending all the ads in a round were shown at the same time. However, in this way we eventually have some more duplicates, since same ads were shown at different hours in the original dataset. Because of this, we carefully build all the rounds so that they do not contain duplicates. We acknowledge that in the case of a user with few non-clicks, those non-clicks can appear multiple times when building rounds and for this reason the algorithms can learn more quickly which arms to avoid. For example, in a borderline case of a user with 19 clicks and 19 non-clicks, we will have 19 rounds with every round containing the same non-clicks.

We build different datasets using the Avazu competition data, by selecting respectively the most frequent 1000 and 10000 users. Selecting an higher number of users would pose serious scalability problems for our algorithm to run. A summary of the datasets obtained is provided in Table 3.2. Because of the preprocessing step, some users are not included in the final datasets. When indeed there are a lot of duplicates, we may not be able to build meaningful rounds for the algorithm as described before. Thus, we discard such users.

We then run all the algorithms listed in section 3.2 on the dataset produced and report results regarding the regret and the running time in section 4.4.

3.7 Software and Platform

Most of the work is done in Python and is available at the following public repository on Github. Different popular libraries for scientific computing, machine learning and data processing are used. In particular, Numpy$^4$ is used for matrix calculations

$^4$http://www.numpy.org
Table 3.2: Description of the 3 final datasets built from Avazu data. "$d$" is the dimension of the contexts, "$k$" the number of arms and "$conj$" denotes the presence of features conjunctions.

<table>
<thead>
<tr>
<th>Users</th>
<th>$d$</th>
<th>$k$</th>
<th>rounds</th>
<th>$conj$</th>
</tr>
</thead>
<tbody>
<tr>
<td>990</td>
<td>30</td>
<td>20</td>
<td>176542</td>
<td>no</td>
</tr>
<tr>
<td>990</td>
<td>60</td>
<td>20</td>
<td>176542</td>
<td>yes</td>
</tr>
<tr>
<td>8667</td>
<td>30</td>
<td>20</td>
<td>407873</td>
<td>no</td>
</tr>
</tbody>
</table>

and other mathematical operations, Matplotlib\(^5\) and Seaborn\(^6\) are used for data visualization and plotting results, while Pandas\(^7\) is used to preprocess raw data and prepare the dataset.

Preprocessing of the Avazu dataset was run on a 32 cores machine provided by the hosting institution while experiments were run on a personal laptop with 4 cores.

3.8 Evaluation

In this section, we describe the two different metrics used to evaluate our algorithm ThompsonCAB.

3.8.1 Regret

In order to evaluate the performance of the different algorithms, the metric used is the regret in the domain of contextual bandits, described in section 2.4. We remind its definition here. Given a decision set $C_t \subseteq \mathbb{R}^d$ and an unknown parameter vector $\theta_x$, the regret can be defined as:

$$R_n = \mathbb{E} \left[ \sum_{t=1}^{n} \max_{x \in C_t} \langle x, \theta_x \rangle - \sum_{t=1}^{n} r_{t,a} \right]$$

In particular, it measures the difference between the total reward gained by a strategy always choosing the best action and the total reward gained by the learner according to its chosen actions.

Depending on the kind of experiment we conduct, in the plots about the results we will show the evolution of the regret over the time horizon, the number of users considered or the number of interactions per user.

Finally, we would like to stress that since all the Thompson sampling algorithms belong to the family of randomized algorithms we need several runs to get reliable estimates of their regret. In particular, we run 5 times each randomized algorithm and average the results.

\(^5\)http://www.matplotlib.org
\(^6\)http://seaborn.pydata.org
\(^7\)http://pandas.pydata.org
3.8.2 Running Time

To evaluate the running time, we use a single machine since the performance can significantly vary across different platforms. In particular, the machine used is a MacBook Pro with a processor 2.7 GHz Intel Core i5.

In this case, not only randomized algorithms are run multiple times but also the deterministic ones. The results are then averaged.
Chapter 4

Results

In this chapter we describe and discuss the experiments we conducted on artificial and real world data.

4.1 Experiment 1: studying the gap parameter $\gamma$

In the first experiment, we want to study the behaviour of the algorithms when changing the gap parameter $\gamma$. To this aim, we use the two variants of ThompCAB described in section 3.1, varying the $\gamma$ parameter in the range $[0, 0.2]$ and compare it to ThompMulti. In particular, by setting $\gamma = 0$ in our algorithm, at the end of each round $t$ only the user served in that round will be updated. We use synthetic data generated as explained previously, with $J = 50$ users, i.e. 10 users per class, and $T = 60$ interactions per user. Results are shown in Figure 4.1 and Figure 4.2 and reported in Table 4.1 and Table 4.2. In particular, in the tables the fields ”Avg upd” and ”Avg neigh”, the average number of users updated every round and the average size of the neighbourhoods are reported.

From the results it is interesting to note that the main advantage of this algorithm comes from the fact that the item chosen at time $t$ is suggested in a collaborative manner. In general, the variant of the algorithm keeping track of the number of times a user has been updated (described in subsection 3.1.3) gives better results. Thus in the rest of the experiments we will only use this better version of our algorithm. On the other hand, this behaviour suggests that updating many users does not seem to help. It should not happen: in the data generated users belonging to the same cluster should benefit from the updates, since they are identical. Thus,

<table>
<thead>
<tr>
<th>Algo</th>
<th>$\gamma$</th>
<th>Regret</th>
<th>Avg upd</th>
<th>Avg neigh</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThompCAB</td>
<td>0</td>
<td>315.06</td>
<td>1</td>
<td>11.63</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.05</td>
<td>355.87</td>
<td>4.48</td>
<td>8.35</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.1</td>
<td>416.92</td>
<td>7.72</td>
<td>8.31</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.2</td>
<td>425.47</td>
<td>9.21</td>
<td>9.17</td>
</tr>
<tr>
<td>ThompMulti</td>
<td>n/a</td>
<td>446.89</td>
<td>1</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 4.1: Table describing the results of the first experiment when using the first variant of the Thompson CAB algorithm.
Figure 4.1: First variant of the algorithm used in the first experiment. Each algorithm is run 10 times and the average is plotted. We do not show the standard deviation by plotting confidence bounds in order to avoid clutter.

Table 4.2: Table describing the results of the first experiment when using the second variant of the Thompson CAB algorithm with different parameters.

<table>
<thead>
<tr>
<th>Algo</th>
<th>( \gamma )</th>
<th>Regret</th>
<th>Avg upd</th>
<th>Avg neigh</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThompCAB</td>
<td>0</td>
<td>316.84</td>
<td>1</td>
<td>11.73</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.05</td>
<td>312.76</td>
<td>1.09</td>
<td>11.13</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.1</td>
<td>327.34</td>
<td>1.54</td>
<td>10.24</td>
</tr>
<tr>
<td>ThompCAB</td>
<td>0.2</td>
<td>368.64</td>
<td>2.78</td>
<td>9.91</td>
</tr>
<tr>
<td>ThompMulti</td>
<td>n/a</td>
<td>442.17</td>
<td>1</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Figure 4.2: Second variant of the algorithm used in the first experiment. Each algorithm is run 10 times and the average is plotted. We do not show the standard deviation by plotting confidence bounds in order to avoid clutter.
it means something does not work sufficiently well: either the algorithm is not able to identify the right users in the clusters or it is including too many users in the clusters. However, the latter issue can be excluded when setting a low parameter $\gamma$. Still, the algorithm is not able to identify all the users belonging to the same class.

### 4.2 Experiment 2: varying the number of interactions per user

In the second experiment we again use synthetic data in order to check what happens when varying the number of interactions per user. In particular, we fixed the number of user to $J = 50$, i.e. 10 per class, and varied the number of interactions $T_u$ from 0 to 100.

Results shown in Figure 4.3 show the cumulative regret vs. the number of interactions per user. In particular, we can see that when the number of interactions per user is low, i.e. $T_u \leq 30$, ThompsonOne is the best. If we do not consider any clustering effect, we need around 60 interactions per user for the ThompsonMulti algorithm to work better. On the other hand, CAB has a slightly lower regret compared to ThompsonCAB. This can be derived from the fact that the users are exactly identical in the model we are using. Thus, a deterministic algorithm like CAB could be favoured compared to its Bayesian counterpart, which introduces randomization when building the neighbourhoods. However, the lower regret of CAB is counterbalanced by its higher running time, as shown in section 4.3.
4.3 Experiment 3: varying the number of users

The third experiment is the last we run on synthetic data. We fix the number of interactions per user $T_u = 50$ and vary the number of users per class from $J = 50$ to $J = 500$, i.e. from 10 to 100 users per class. We show the average cumulative per-user regret vs. the number of users in Figure 4.4.

In general, the average regret per user seems to follow a fixed trend, thus showing no changes in the expected behaviour as the number of users grows. As expected, ThompMulti is not able to do better than ThompsonOne, since we have less than 60 interactions per user. Also, CAB performs better than ThompCAB. Again, this can be explained with the fact that a deterministic algorithm should perform better when we have identical users. Fluctuations in the performances are caused by the random generation of data at disposal.

On the other hand, when considering the running time CAB runs much slower as the number of users grows, as reported in Table 4.3 and shown in Figure 4.5.
Figure 4.5: Running time of the algorithms: on the x-axis the number of users considered, on the y-axis the running time in seconds.

Table 4.3: Table describing the running time of the algorithms in seconds when varying the number of users.
4.4 Experiment 4: bandits on real world data

The fourth experiment consists in running different bandit algorithms on real world data. In general, performances are worse compared to the case of artificial data, since real data contains much more "noise". Since it is difficult to distinguish between different algorithms when plotting the cumulative regret \( R_T \), we decided to plot instead \( R_T / T \). Ideally, this ratio should go to 0 as \( T \to \infty \). However, as shown from results this is not the case.

On all the 3 datasets we used (described in Table 3.2), LinUcbOne and ThompsonOne algorithms have an initial fast convergence to a fixed value after which they do not improve much, as we can see in Figure 4.6, Figure 4.7 and Figure 4.8. On the other hand, the general trend for algorithms having a single bandit instance per user is to improve over time, as expected. Furthermore, algorithms clustering users (i.e. CAB, ThompCAB) learn faster compared to those which consider independent bandit instances for users (i.e. ThompMulti, LinUcbMulti). Overall, when considering the final result about \( R_T / T \), the single instances are better when considering 1000 users, thus meaning that we would probably need more interactions per user for the other algorithms to perform well. As shown in experiments with artificial data, we need a certain amount of interactions per user for the algorithms considering one bandit instance per user to be better. Also, the use of feature conjunctions helps all the algorithm in getting a lower regret over time, as shown in Table 4.5 and Figure 4.7. On the other hand, when we consider the bigger dataset, LinUcbOne and ThompsonOne seem to be worse. This may be explained as follows: until we have a limited number of users a single bandit instance may be able to generalize well. When instead the number of users grows, it may be more difficult for these algorithm to generalize, while having multiple instances (with clustering) might be a better alternative. Finally, when considering the running time, from Table 4.4 and Table 4.6 we can clearly see that ThompCAB is faster than CAB, but both are much slower compared to the other algorithms.
Figure 4.6: Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 1000 users. In this case features conjunctions are not used.

<table>
<thead>
<tr>
<th>Algo</th>
<th>$\alpha$, $v$</th>
<th>$\gamma$</th>
<th>$R_T/T$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThompCAB</td>
<td>0.01, 0.1</td>
<td>0.78</td>
<td>758.6</td>
<td></td>
</tr>
<tr>
<td>CAB</td>
<td>0.01, 0.1</td>
<td>0.77</td>
<td>2560.2</td>
<td></td>
</tr>
<tr>
<td>ThompMulti</td>
<td>0.001, n/a</td>
<td>0.81</td>
<td>62.7</td>
<td></td>
</tr>
<tr>
<td>LinUcbMulti</td>
<td>0.1, n/a</td>
<td>0.83</td>
<td>40.4</td>
<td></td>
</tr>
<tr>
<td>ThompsonOne</td>
<td>0.1, n/a</td>
<td>0.79</td>
<td>63.1</td>
<td></td>
</tr>
<tr>
<td>LinUcbOne</td>
<td>0.01, n/a</td>
<td>0.77</td>
<td>33.59</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Results on the dataset built with the most frequent 1000 users. In this case features conjunctions are not used.
Figure 4.7: Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 1000 users. In this case features conjunctions are used.

<table>
<thead>
<tr>
<th>Algo</th>
<th>$\alpha, v$</th>
<th>$\gamma$</th>
<th>$R_T/T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThompCAB</td>
<td>0.01</td>
<td>0.1</td>
<td>0.74</td>
</tr>
<tr>
<td>CAB</td>
<td>0.01</td>
<td>0.1</td>
<td>0.75</td>
</tr>
<tr>
<td>ThompMulti</td>
<td>0.01</td>
<td>n/a</td>
<td>0.80</td>
</tr>
<tr>
<td>LinUcbMulti</td>
<td>0.001</td>
<td>n/a</td>
<td>0.83</td>
</tr>
<tr>
<td>ThompsonOne</td>
<td>0.01</td>
<td>n/a</td>
<td>0.73</td>
</tr>
<tr>
<td>LinUcbOne</td>
<td>0.001</td>
<td>n/a</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table 4.5: Results on the dataset built with the most frequent 1000 users. In this case features conjunctions are used.
Figure 4.8: Plot showing $R_T/T$ for different algorithms when using the dataset built with the most frequent 10000 users.

<table>
<thead>
<tr>
<th>Algo</th>
<th>$\alpha, \gamma$</th>
<th>$R_T/T$</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThompCAB</td>
<td>0.01 0.05</td>
<td>0.841</td>
<td>6352</td>
</tr>
<tr>
<td>CAB</td>
<td>0.01 0.05</td>
<td>0.859</td>
<td>11333</td>
</tr>
<tr>
<td>ThompMulti</td>
<td>0.001 n/a</td>
<td>0.874</td>
<td>149</td>
</tr>
<tr>
<td>LinUcbMulti</td>
<td>0.1 n/a</td>
<td>0.879</td>
<td>83</td>
</tr>
<tr>
<td>ThompsonOne</td>
<td>0.1 n/a</td>
<td>0.864</td>
<td>168</td>
</tr>
<tr>
<td>LinUcbOne</td>
<td>0.01 n/a</td>
<td>0.864</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 4.6: Results on the dataset built with the most frequent 10000 users. In this case the probability of sampling users when building the neighbourhoods was $p = 0.1$ for both CAB and ThompCAB.
Chapter 5

Conclusions

5.1 Discussion and future work

Empirical results of the experiments conducted reveal that our algorithm has a regret comparable to CAB (its UCB version), but when it comes to running time it is much faster. However, when running the algorithm on synthetic data our algorithm is not able to identify very well users belonging to the same cluster (as shown in section 4.1). This indicates that there might be something wrong when building neighbourhoods, which are based on a Bayesian version of confidence bound. To fully understand the validity of this principle a thorough theoretical analysis should be conducted. Such an analysis should also provide upper and lower bounds on the regret of our algorithm. We expect the theoretical analysis to not be very different compared to the one conducted in [Gentile et al., 2017], i.e. the regret should scale with the size of clusters built (see subsection 2.5.1).

On the other hand, when using our algorithm on real-world data the results are more difficult to evaluate: we consider the setting of personalized recommendations and for our algorithms to be effective we need a lot of interactions per user. When it comes to real-world data, it is difficult to have such a scenario, since most users only have a few interactions with a recommender system. In the experiment conducted (see section 4.4), our algorithm gave results in line with CAB (or better with the bigger dataset) in terms of regret. Also, the impression is that when using bigger datasets, i.e. with a lot of users, individual bandit instances for each user tends to be a better solution, since a unique bandit instance may not be able to generalize well if the users are very different among themselves, which is likely to happen as their number grows. However, when the number becomes large (i.e. > 1000), CAB and its Bayesian version are not directly employable because of their big running time compared to other algorithms (see section 4.3 and especially section 4.4). This issue prevented us from experimenting with even bigger datasets and is a severe limitation, since in a real-world system the number of users would likely be huge. We tried different ways to overcome this issue, such as optimizing the code and subsampling users. There are mainly two bottlenecks in our algorithm: the sampling of the users’ parameters \( \hat{\theta} \) and the update subroutine of the users in the clusters. Both of these processes are independent among users, thus a parallel implementation should make things faster. We did not try this possibility, which is left for future work.

There are also other directions for our work to evolve. Other ideas to make the algorithms (both CAB and ThompCAB) more scalable involve for example to avoid the
computation of clusters at each iteration: we could compute them every $n$ iterations, where $n$ is a fixed variable for any given user. Also, the number of contexts, i.e. items, is not infinite and the idea of clustering items could be applied as well, as done for example in [Li et al., 2016].

Another problem we encountered was the choice of hyperparameters. Our strategy was based on a grid search: given a set of values we tried all of them in different combinations. Besides being very time consuming, this solution might not be the safest strategy to decide the hyperparameters, since our set of values is decided beforehand and if the best combination of values is not contained in that set then it will be missed. Other methods to decide the values of the hyperparameters could be based on Bayesian optimization, as explained for example in [Snoek et al., 2012].

Finally, in the evaluation method it would be interesting to test our algorithm on other real-world datasets, where the features are possibly not anonymized.

5.2 A view on ethics and society

Nowadays, recommender systems are pervasive in the existing connected society. Offering better recommendations can provide a more pleasant experience to the users of every-day services, (e.g. movies to watch, places to visit, new friends on social networks and so on) and make them spend less time to decide what option to pick. At the same time it is in the interest of service providers to give proper recommendations on products in order to keep clients and improve their revenues. With our work we tried to address these problems by developing an algorithm which should give accurate recommendations when considering a fully personalized recommender system, i.e. each user is treated differently from the others. However, in order to learn faster users’ tastes we try to cluster those who are similar. Fundamentally, we think that our algorithm works better when a certain number of interactions per user are present. Thus, we assume a certain amount of users’ data will be collected. On the other hand, we did not address specifically problems related to ethics, but we are aware that this kind of algorithms when deployed in real-world often work better when more data about users is collected. Thus, there is a trade-off between the accuracy of recommendations and the extent to which users are willing to release information about their own preferences. It would be interesting to conduct studies on how information losses can affect the quality of recommendations. However, such a study should be conducted in the private sector, since private companies are those which collect large amount of data (and almost always are not willing to share it with the public).
Bibliography


