Ad optimisation with a collaborative multi-armed bandit algorithm

Måns Rönnberg
Abstract

This report describes the implementation of a collaborative multi-armed bandit algorithm in Python. The algorithm is tested on different online ad campaign data with the purpose of finding out if a collaborative bandit can come to an accurate decision faster on which ad in a set of ads is the best performer with regards to click-through rate. If the best performing ad can be found quickly, that ad can be displayed the most, thus maximizing profits for the advertiser. Analysis is also done on the ad data to see if similar ads perform similarly, which is important for the idea to work. The experiments performed show a decrease in performance when using collaboration between ads compared to no collaboration over a longer time period, however a slight increase in the short term. Some issues with the work and possible future extensions are also discussed.
# Contents

1 Introduction 3

1.1 Problem description ......................... 3
1.2 Report outline ............................. 4

2 Background and related work 5

2.1 The company ............................... 5
2.2 Multi-armed bandit algorithms ............... 5
2.3 Collaborative filtering ....................... 6

3 Method and Implementation 8

3.1 The data set ................................ 8
3.2 Extracting relevant data ..................... 8
3.3 Prototype implementation .................... 9
3.4 How the algorithm works ..................... 13

4 Results and discussion 14

4.1 The first ad campaign ....................... 14
4.2 The second ad campaign ..................... 15
Introduction

1.1 Problem description

With the number of internet users rapidly increasing each year [2], the market of internet display advertising is also growing. In 2019, the internet advertising market was a 319 billion US dollar market and it is projected to reach 1,089 billion by 2027 [7]. Internet advertisers have a lot to gain by displaying effective ads, usually measured by click-through rate (CTR). They want to generate as many clicks as possible as cheap as possible.

When it comes to internet display advertising, advertisers often create a large number of ads with different properties for different devices, regions or target groups. These include different languages, ad sizes, dimensions, colors, texts and images. These factors can have a large impact on how well the ad performs, that is, how often the ad is clicked on by users being exposed to it[3]. This is known as the ad’s click-through rate. The click-through rate is calculated by taking the amount of clicks an ad has gotten divided by the amount of impressions (views). An important problem for the advertiser is to quickly figure out which ad is the best performer among a group of ads and then exploit that information and display the best ad the most. For this purpose, multi-armed bandit algorithms are useful. The name multi-armed bandit comes from the classic one armed bandit slot machines present in many casinos. If multiple one-armed bandits are present, a gambler might be interested in finding out which of these machines will give the highest average pay-out. This is an optimization problem. The gambler wants, as quickly as possible, find the best machine and then only gamble on that specific machine to maximize their earnings.
With internet display advertising, the advertiser is looking for a similar thing: which ad, in a set of ads, is performing the best? To figure that out, the advertiser will often try displaying all of the ads to many different users and draw a conclusion on which ad was the most popular by looking at their click-through rates[1].

Coming to a conclusion is usually not very difficult if the impression rate is high enough. With many thousands or millions of ad impressions, it is often clear to see which ad performs the best using a traditional multi-armed bandit algorithm or A/B testing. If, however, a certain set of ads for some reason has a very low impression rate, it can be hard to reach a confident decision in a reasonable amount of time. An interesting problem is therefore to see if a collaborative algorithm can be used to come to a conclusion faster. A collaborative algorithm would take ads that are similar, but differ slightly for example in dimensions or language, and in some way combine information on their click through rates. This is where the focus of this thesis will lie.

1.2 Report outline

The report is organized into four parts. The background section covers what multi-armed bandits are and how they work and other necessary background information. The method and implementation section explains the data set being analyzed, how the analysis of the data was done, the software being used and more. An implemented collaborative bandit prototype is also described here. Finally, the results and discussion section covers the results of this thesis project and discusses potential reasons and future implications.
Background and related work

2.1 The company

This thesis is written at the Stockholm-based company Bannerflow. Bannerflow has a platform where their customers can design and publish banner ads online and it is the data from this platform that the thesis builds upon.

2.2 Multi-armed bandit algorithms

There exist several multi-armed bandit algorithms, with different levels of complexity. A simple greedy approach to a multi-armed bandit tries all different arms once, sees which arm gave the best outcome and then continues pulling that arm forever. This approach has obvious flaws. If the probability distribution curves for two arms are uneven and one arm has much higher probability of giving a more profitable result, the result can still be unlucky and the good arm give a bad result. If the arm with lower probability of giving a profitable result at the same time gets lucky and gives a good result, the greedy algorithm would think that the bad arm was the good one and vice versa. This would be very unfortunate since the actual bad arm would be continued to be pulled forever and the actual good arm would never be tried again, giving a worse result in the long run.

To combat this issue, many solutions have been proposed. One of these include an extra parameter in the equation, $\epsilon$, which the algorithm designer sets manually. With probability $1 - \epsilon$, after all arms have been tried once,
a random arm is pulled instead of the arm with the current best average result. This allows the algorithm to still try other arms even if a bad arm got lucky at first. Using this method, given enough time, the actual best arm will be found. This method has one major disadvantage, and that is that the algorithm will keep trying other arms with the same probability forever. If a large amount of time and arm pulls have already been done, it will already be quite clear which arm is the best and the value of trying other arms will go down. To keep trying other arms is then not beneficial to maximizing the total outcome, at that point it would be better to simply keep pulling the one with the best average outcome [9].

A third method is an algorithm called The Upper Confidence Bound (UCB) Algorithm. This algorithm is similar to the greedy $\epsilon$ algorithm, but instead of having $\epsilon$ statically set from the beginning, the probability of trying other arms versus picking the currently best one adapts during the time the algorithm is running. This improves the total outcome in the long run by not attempting as many other arms as time and the number of total pulls increases. Equation 2.1 shows how the math behind the USB algorithm works.

$$A_t = \arg \max_a \left[ Q_t(a) + c \sqrt{\frac{\log t}{N_t(a)}} \right]$$

Where $A_t$ is the action taken at time step $t$, $Q_t(a)$ is the estimated value of action $a$ at time step $t$, $N_t(a)$ is the number of times action $a$ has been taken prior to time step $t$, $c$ is the confidence value which controls the level of exploration [8].

UCB chooses between exploration (trying different arms) and exploitation (pulling the arm with the current maximum average result) in an effective manner close to the optimal solution if the best bandit was given from the start. It is therefore the algorithm implemented on the data this thesis analyzes and the algorithm that will be expanded upon.

### 2.3 Collaborative filtering

Collaborative filtering is a method of automatically predicting a user’s preferences by looking at other similar users. The idea behind collaborative
Collaborative filtering is the general idea that the collaborative part of the multi-armed bandit in this thesis will be built on. If one ad has a certain popularity among a user base, another similar ad should have a similar popularity.
Method and Implementation

3.1 The data set

The data set being analyzed is a sample set of The Company’s own massive data warehouse. The data warehouse contains a very large number (roughly 200 billion rows at the time of writing) of ads and statistics about how they perform among other things. Information such as time, location and whether or not the viewer of the ad is likely to be a bot are included.

The data warehouse is collecting information about the ads in "buckets", or time intervals. That means that the warehouse is updated once per hour with all the information about the ads being displayed in that hour.

3.2 Extracting relevant data

Due to the sheer size of this data warehouse, it was not feasible to attempt to analyze all of it. Two different ad campaigns spanning a few months each, with ad designs that were fitting for analysis were selected. Some of these ads had similar design but with differing sizes or proportions. These kinds of ads were deemed to be fitting as it is here a collaborative multi-armed bandit would be most useful. Ads that are very different from each other would not benefit from a collaborative multi-armed bandit since they need a certain degree of similarity for the collaborative bandit to be effective.

Extraction from the data warehouse was done with regular SQL queries.
The data could then be saved to a csv format to be more easily analyzed in Python.

### 3.3 Prototype implementation

A collaborative bandit prototype was implemented in Python. This bandit builds upon an already existing UCB algorithm [4], but also takes collaborative features into consideration when calculating the reward for each arm. At first the idea was that the algorithm itself would determine which ads in the data set were similar to others by looking at attributes such as size, dimensions and language. This method, however, proved to be quite poor at finding actual similar ads. A major problem was that ads could have very different design despite seeming similar when looking at their attributes in the data set. They could for example have completely different design or color schemes, differences that were not noticeable without visual inspection of the ads. Another issue with automatically detecting similar ads was that it was resource intensive to do the extra calculations at each step, especially to find ads with similar proportions which requires division. To combat this issue it was decided that the user would manually enter in which ads in the data set that were competing and also which ads were similar to one another. This simplifies the algorithm and would also not affect the user much as it would be quick to fill in which ads were similar since the pool of ads being tested usually is not very large.

Figure 3.1 and 3.2 show two different ads in the same ad campaign that could and were deemed similar due to their very similar design, message and proportions. The only difference between these two ads is that the first one is a bit wider. It is therefore reasonable to assume that these ads would perform similarly and could be used collaboratively in a multi-armed bandit.

Of course not all ad campaigns have ads that are this similar. The advertiser may still decide to put ads that are not exactly the same as similar to boost the impression rate in the algorithm. The ad banners in figure 3.3 and in 3.4 are quite similar to each other but there is a bigger difference than between 3.1 and 3.2. There is however a larger difference between these two and the previous two so it would be reasonable for an advertiser using the collaborative bandit algorithm to put 3.3 and 3.4 as similar ads, if the individual impression rates are low. The trade off is between speed and precision of the algorithm.
Figure 3.1: Ad banner with dimensions 909x250 px

Figure 3.2: Ad banner with dimensions 659x250 px
Figure 3.3: Ad banner with dimensions 250x500 px
Figure 3.4: Ad banner with dimensions 551x735 px
### 3.4 How the algorithm works

The algorithm works by first reading the data set csv file containing the ad data (size, impressions, clicks etc). Since the data is batched, the algorithm must read each line for every time step. For each time step, information on how the different ads performed during that time is available. Since the algorithm is collaborative, information for similar ads are also taken into consideration. As mentioned, this is implemented so that the user manually enters which ads they consider similar before starting the algorithm.

When the algorithm is running, the impressions and clicks of the similar ads are added together with the ads that are competing against each other to calculate their respective upper confidence bound. The upper confidence bound for each competing ad is calculated at each time step by adding the average click-through rate of previous selections together with a delta (uncertainty) value that increases as time goes on but decreases for each impression. The competing ad with the highest confidence bound is selected as the winner of that specific time step and its impressions and clicks are registered for future calculations. The data for the ads that were not selected is not saved since the idea is that the algorithm should work in real time and only display one competing ad at a time.

Other than that, the algorithm works like any UCB algorithm with non-batched data and not taking collaborative features into consideration. The full source code for the prototype implementation is available in the appendix.
Results and discussion

4.1 The first ad campaign

Running the algorithm on the two ads in figure 3.1 and figure 3.3 without any collaboration and with \( c = 0.01 \) gives the following result after the total 694 time steps in the data set: The ad in figure 3.1 was selected 0.203661 of the time and the ad in figure 3.3 was selected 0.796339 of the time. The total impressions of the selected ads were 26986 and the total clicks 60, resulting in a final click-through rate of 0.002223.

Running the algorithm with the same two ads but with collaboration with other, similar ads in the data set (three similar ads for figure 3.1 and two similar ads for figure 3.3) gives the following result after 694 time steps: The ad in figure 3.1 was selected 0.243699 of the time and the ad in figure 3.3 was selected 0.756301 of the time. The total impressions of the selected ads were 35990 and the total clicks 56, resulting in a final click-through rate of 0.001556.

Looking at the performance of each of the two competing ads over the total time period shows that ad 3.1 had 69690 impressions and 44 clicks giving a click-through rate of 0.000631. 3.3 had 23776 impressions and 61 clicks, a click-through rate of 0.002566.

The ads similar to 3.1 had a total of 259356 impressions and 221 clicks, resulting in a click-through rate of 0.000852.

The ads similar to 3.3 had a total of 307414 impressions and 311 clicks, resulting in a click-through rate of 0.001012.
When only the first 50 time steps are taken into consideration, the click-through rate is 0.009276 without collaboration and 0.009653 with collaboration.

### 4.2 The second ad campaign

The second ad campaign had three ads that were selected to compete against each other. These ads had different messages and appearance.

Running the algorithm with $c = 0.01$ and no collaboration between similar ads in the campaign gave the following result after the total 427 time steps in this data set: The ad in figure 4.1 was selected 0.047398 of the time, the ad...
Figure 4.2: Ad banner from second campaign with dimensions 320x160 px

Figure 4.3: Ad banner from second campaign with dimensions 980x240 px

in 4.2 was selected 0.817985 of the time and the ad in figure 4.3 was selected 0.134617 of the time. The total impressions of the selected ads were 12933 and the total clicks 238, resulting in a final click-through rate of 0.018403.

Running the algorithm on the same ads but with collaboration between other similar ads in the campaign (two similar ads for 4.1, three similar for 4.2 and two similar for 4.3) gives the following result after the 647 time steps: The ad in figure 4.1 was selected 0.110323 of the time, the ad in figure 4.2 was selected 0.813092 of the time and the ad in figure 4.3 was selected 0.076585 of the time. The total impressions of the selected ads were 32639 and the total clicks 250, resulting in a final click-through rate of 0.007660.

Looking at the individual performance of the three competing ads over the entire time period shows that the ad in figure 4.1 had 47167 impressions and 266 clicks, resulting in a click-through rate of 0.005640. The ad in figure 4.2 had 10746 impressions and 228 clicks resulting in a click-through rate of 0.021217. The ad in figure 4.3 had 158506 impressions and 388 clicks resulting in a click-through rate of 0.002448.

The ads similar to 4.1 had a total of 705 clicks and 261963 impressions, a
click-through rate of 0.002691.

The ads similar to 4.2 had a total of 1553 clicks and 920204 impressions, a click-through rate of 0.001688.

The ads similar to 4.3 had 363 clicks and 350349 impressions, a click-through rate of 0.001036.

When only looking at the first 50 time steps, the results look different. The non-collaborative gets 70 clicks and 2599 impressions, a click-through rate of 0.026933, and the collaborative version gets 58 clicks for 1835 impressions, a click-through rate of 0.031608.

4.3 Results discussion and conclusions

The results confirm the initial theory that similar looking ads also perform similarly in regards to click-through rate. This suggests that even though the advantage of a collaborative bandit could not be proven with the prototype algorithm and the available data, a bandit should be able to take advantage of collaboration to find the best performing ad quicker with greater confidence.

Both ad campaigns show a slight improvement of the click-through rate in the collaborative bandit compared to the non-collaborative when only looking at the first 50 time steps, but worse performance when using the entire data set. There could be different reasons for this, but one reason could be that the algorithm benefits from more data in the beginning but in the long run the extra data makes the algorithm less precise, locking in on a group of ads and not the exact ad with the best performance. This suggests that a collaborative bandit could work better initially but a non-collaborative algorithm would work better in the long term.

It is important to note that results will vary depending on the ad data. If similar ads have very different click through rates, in contrast to the assumption that they should perform similarly, the collaborative bandit may not find the optimal ad faster than a regular bandit.

The $c$, or exploration, parameter in the algorithm has a large impact on the result. In the tests conducted in this thesis the higher the value of $c$ the worse the result became. This is because one of the ads had much better
performance on most time steps, throughout the entire time period. If the popularity of the ads had shifted more over the entire course of the data set, the exploration becomes much more important. This is true both for collaborative and non-collaborative bandits.

4.4 Problems with the data

One issue with the data provided was that it was often “uneven”. Certain time steps one ad might have hundreds of impressions while another barely had any. This was problematic since in a regular non-batched upper confidence bound algorithm, exactly one new impression should be made every time step. This meant that different amount of new data were being evaluated each time step. In an ideal situation, each ad in the test data set would have the same number of impressions every time step, for example 100.

This issue was however automatically lessened bit when many similar ads were taken into consideration, since a specific ad’s performance that time step did not have as much impact then. The amount of time steps with 0 impressions decreased a lot.

One problem with this analysis of the data was that it was difficult and manually intensive to extract data from the data warehouse. The big reason for this was that the ads and campaigns had to be manually inspected to see if they were similar in design or not. If there was somehow an automatic way of finding suitable ad campaigns with similar ads in them, more tests could be performed and the result would be more certain.
Future work

There are many ways this work can be extended. Although difficult, an automatic way of detecting similar ads, like mentioned previously in the report, could be implemented. This would save the user of the algorithm some time however its effectiveness would depend on the accuracy of such an automatic similar ad detector.

Another way the algorithm could potentially be improved would be to implement weights for how similar different ads are. Ads more similar to a competing ad could have a higher weight and ads less similar could have a lower weight. This would mean that more ads could be included in the similar ads set when running the algorithm and could perhaps lead to a quicker or better result.

A third way the work could be improved would be to analyze and use more data and different ad campaigns to test the algorithm on for a more confident result. It is possible that the results will be different when other data sets are used.
import csv
import math
from re import S
import pandas as pd
import sys

numbers_of_selections = {}
sums_of_clicks = {}
sums_of_impressions = {}
timestep = 0
clicks_this_timestep = {}
impressions_this_timestep = {}
real_current_impressions_this_timestep = {}
real_current_clicks_this_timestep = {}
total_reward = 0
total_impressions = 0
ads_selected = []
impression_stats = []

#IDs of competing ads
competitor1 = ""
...
competitorN = ""

list_of_competitors = [competitor1, ..., competitorN]

# DataFrame to read input CS file
dataFrame = pd.read_csv("dataset.csv")

# sorting the data set by date
dataFrame.sort_values(["DATEHOUR"], axis=0, ascending=True, inplace=True, na_position='first')
data = DataFrame.values.tolist()
similar_ads = {  # for collaboration
    competitor1: ['"", ... , ""],
    ...
    competitorN: ['"", ... , ""
}

prev_time = data[0][0]
for row in data:
    real_competitor = False
    current_competitor = ""

    for key, value in similar_ads.items():
        if str(row[3]) == key:
            real_competitor = True
            current_competitor = key
            break
        elif str(row[3]) in value:  # collaborative
            current_competitor = key
            break
    if current_competitor == "":
        continue

    current_time = row[0]
    if current_time != prev_time:  # new timestep
        ad = 0
        max_upper_bound = 0
        for i in list_of_competitors:
            if ads_selected.count(i) > 0:
                average_reward = sums_of_clicks.get(i, 0) / (sums_of_impressions.get(i, 0) + 1)
                delta_i = 0.01 * math.sqrt(math.log(total_impressions+1) / (sums_of_impressions.get(i, 0) + 1))
                # exploration term, increase c for more exploration
            else:  # if ad has never been shown
                upper_bound = 1e400
                if upper_bound > max_upper_bound:  # Display ad with highest upper bound
                    max_upper_bound = upper_bound
                    ad = i
                ads_selected.append(ad)

                impressions = impressions_this_timestep.get(ad, 0)
                sums_of_impressions[ad] = sums_of_impressions.get(ad, 0) + impressions
                total_impressions +=
                real_current_impressions_this_timestep.get(ad, 0) #
impressions

    impression_stats += impressions * [ad]
    reward = clicks_this_timestep.get(ad, 0)
    sums_of_clicks[ad] = sums_of_clicks.get(ad, 0) + reward
    total_reward += real_current_clicks_this_timestep.get(ad, 0)# reward

    timestep += 1
    prev_time = current_time

    impressions_this_timestep = dict.fromkeys(
        impressions_this_timestep, 0) # resetting
    clicks_this_timestep = dict.fromkeys(
        clicks_this_timestep, 0)
    real_current_impressions_this_timestep = dict.
        fromkeys(real_current_impressions_this_timestep, 0)
    real_current_clicks_this_timestep = dict.fromkeys(
        real_current_clicks_this_timestep, 0)

    if(row[32] == 1):  # if TYPE is impression
        impressions_this_timestep[current_competitor] =
        impressions_this_timestep[current_competitor].get(
            current_competitor, 0) + row[35]
        if real_competitor: # if actual competitor, not similar ad
            real_current_impressions_this_timestep[
                current_competitor] =
            real_current_impressions_this_timestep.get(
                current_competitor, 0) + row[35]
    elif (row[32] == 2): # if TYPE is click
        clicks_this_timestep[current_competitor] =
        clicks_this_timestep.get(current_competitor, 0) + row[35]
        if real_competitor:
            real_current_clicks_this_timestep[
                current_competitor] = real_current_clicks_this_timestep.
                get(current_competitor, 0) + row[35]

    ts = pd.Series(impression_stats, dtype='float64').
        value_counts(normalize = True)
    print(ts)
    print("total reward: ", total_reward)
    print("total impressions: ", total_impressions)
    print("CTR: ", total_reward/total_impressions)

Listing A.1: Python code
Bibliography


