Streaming Course

Master IASD

Suspicious activities project

Authors:
Maryline Chen
Emilie Chhean

July 6, 2021

Contents

1 Introduction

For retargeting companies that provide online display advertisements like *Criteo*, it is important to identify what are the valid clicks. In the context of this project, we will seek to detect some of those suspicious activities by creating a Flink application which will read from Kafka clicks and displays. From these two topics, we will extract the fraudulent patterns that have been simulated and store the results in an output file.

We can distinguish this project in two parts: offline analysis and online implementation. For simplicity, it was recommended to do offline analysis which indeed proved to be more efficient. Then in a second step, we implemented a Flink application on Scala in order to manage online data and detect fraudulent behavior in real time.

2 Project environment and execution

We wanted at first to explore the data on Scala but not being familiar with this language and as recommended to detect outliers, we finally chose to do an offline analysis before.

To do so, we collected the data streams in real time by directly reading and writing them into a text file. We then investigated in Python language via a Google Colab notebook to make it easier for both of us to access after each change. To run our offline analysis, one can execute the notebook *offline_analysis.ipynb* or visualise it through this *Github repository*.

After doing the offline analysis, we can now turn to the online part :

First, one has to run the docker-compose file *docker-compose.yml* by executing ***docker-compose rm -f; docker-compose up*** in the same directory as the file.

Then, one can execute *StreamingJob.scala* to launch our fraudulent activities detector to retrieve the patterns and its outputs. Two output files will be generated *suspiciousUID.txt* and *suspiciousIP.txt* - the meaning of these files are explained in the following sections. An example of them is given in *suspiciousInputs repository*.

It has been done in Scala and on IntelliJ IDE, that can be found in this Github via the repository *quickstart*.

3 Offline Analysis

3.1 Preprocessing

3.1.1 Load data

We first loaded data and displayed the *clicks* and *displays* in two dataframes to explore them before doing any analysis. We can actually put them in the same dataframe but we preferred to keep one per topic. For both of them, we have the same features:

- eventType : *clicks* or *displays* - depending on the studied dataframe.
- uid
- timestamp
- ip
- impressionId

We also added a new feature **date** to convert the timestamp from a string format as "1623489620" to a datetime format as "2021-06-12 09:20:20", in order to be able to filter on a time window.

3.1.2 General information

The informations about the clicks and the displays dataframes are accounted in this following table:

	number of rows	unique UIDs	unique IPs	unique impressionIDs
clicks	112 213	43 923	10 000	112 213
displays	330 254	155 983	10 001	330 254

We observed for both of our offline datasets, there are as many impressionId as there are rows.

3.1.3 Filter on a given window

Firstly, to choose a window that is representative of all our offline data, the Figure 1 shows the clicks and displays distribution over time.

We implemented a function to filter a given dataframe over a time window. This function takes in parameters the dataframe that we want to filter, a start time and a window size. To check that the filtered data falls within the time window we chose and that this window is representative of the entire dataset, the Figure 2 counts the number of displays and clicks for each minute of the time window.

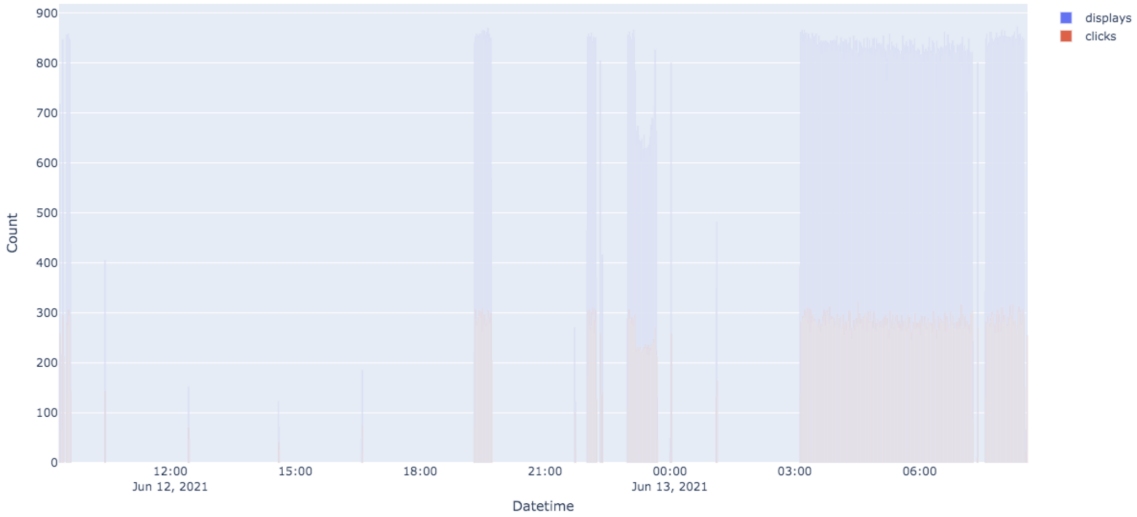

Figure 1: Distribution of clicks and displays over time

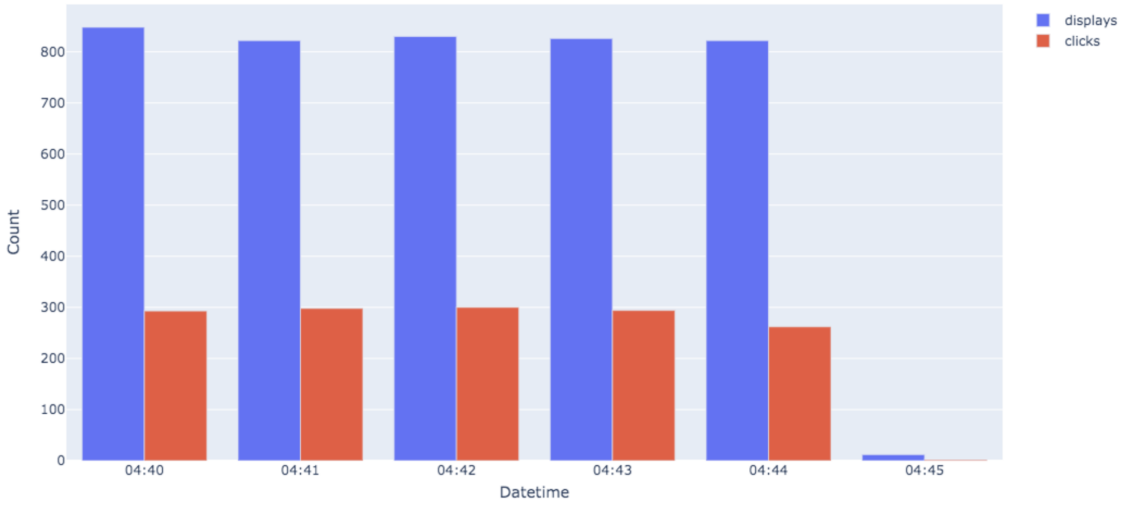

Figure 2: Distribution of clicks and displays in a given time window. Inclusive time window of 5 minutes thus giving 6 bars.

3.2 Suspicious activities and its outputs

We finally detected 4 patterns, 3 of which concern potential fraudulent UIDs and the last one concerns potential fraudulent IPs. Three patterns were requested, but we retained four that seemed the most intuitive.

The **Click Through Rate** (CTR) corresponds to the number of clicks registered by your ad, divided by the number of times it was displayed and is expressed in percents (clicks $\div$ impressions $=$ CTR). For example, if you get 5 clicks and 100 impressions, your CTR is 5%.

We had as hint that a normal CTR is around 10%. The intuition would then be to look at this hint and if one of our patterns is indeed a fraudulent pattern then it should help to find a normal CTR.

For each of the patterns, there is associated threshold(s) which can be varied (e.g. choose a threshold more appropriate to business needs). In the notebook, we defined the *time_window* equal to 5.

3.2.1 Pattern 1 : UIDs with a Click Through Rate too high

Starting from the clue given, this led us to focus on the CTR, more precisely the *CTR per UID*. This pattern is detected if for a given UID, its click through rate is higher than a certain percentage. The idea here is that we would like to know which users are clicking far too often for a given minimum number of displays.

To do this, we merged clicks and displays dataframes, and we define some thresholds to decide whether a UID is suspicious :

- **min_ctr** : set at 0.3, having more than this CTR threshold could be suspicious (this value can be changed)
- **min_displays_ctr** : set at $(3 * time\_window)$, to ensure that we won't consider suspicious cases like $1click/1display$ or $2clicks/2displays$.

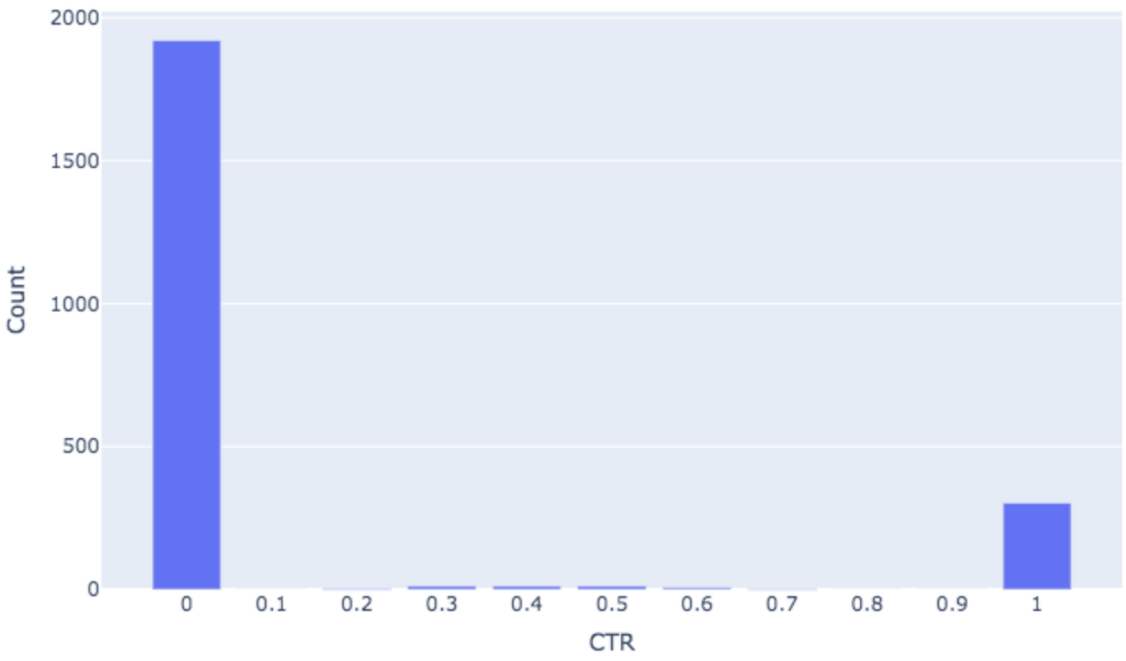

Figure 3: UID counter per CTR

In the Figure 3, it can be observed that two different CTR stand out. Most of the CTR are equal to 0, with about 1900 UIDs. The second highest CTR is 1 with about 300 UIDs.

Finally, taking all UIDs with a CTR greater than *min_ctr* and a number of displays at least greater than *min_displays_ctr*, we found 46 UIDs that are suspicious.

3.2.2 Pattern 2 : UIDs that are associated with more than one IP

Still on the idea of detecting suspicious users, this pattern is identified if for a given UID, its number of unique IP is higher than 2 (included). Indeed, we considered a UID to be fraudulent if it is associated with more than one IP.

Finally, there are 56 UIDs that are associated with more than one IP.

3.2.3 Pattern 3 : UIDs where we have too many displays/clicks

The third pattern concerns UIDs that have too many displays. By lots of displays we mean a lot of *impressionID*. The idea here is that just looking at CTR was not enough, as it allows mainly focus on clicks over displays. However, there could be fraudulent behaviour that would deliberately generate displays, hence this pattern.

We also set a threshold for a minimum number of displays/clicks equal to $(4 * time\_window)$.

For this pattern, we found 50 suspicious UIDs that have too many displays on a fix window.

3.2.4 Pattern 4 : IPs that are associated with too many UID

This last pattern concerns the detection of suspicious IPs. It detects IPs that are associated with too many UIDs. Indeed, if for a given IP, its number of unique UIDs is too high, we consider it as fraudulent.

For this pattern, we set a threshold for a minimum number of UIDs equal to $(2 * time\_window)$.

We eventually found 1 suspicious IP that is associated with around 300 UIDs in our time window.

3.3 Ensuring the reliability of our patterns

Finally, the suspicious UIDs entail 2.21% as shown in the Figure 4, or 56 UIDs. To ensure that our patterns are correct, we compute the CTR after removing all suspicious UIDs and IPs.

Before removing suspicious, over the 5 minutes window, when we take all clicks/displays, the global CTR is equal to 34.83%. After removing suspicious, the CTR is then equal to 13.82%. Thus the global CTR has decreased to almost 10%, and we get closer to a normal CTR.

We then wanted to see if there was a pattern that was more discriminating than the others. The Figure 5 shows the distribution of suspicious UIDs over patterns.

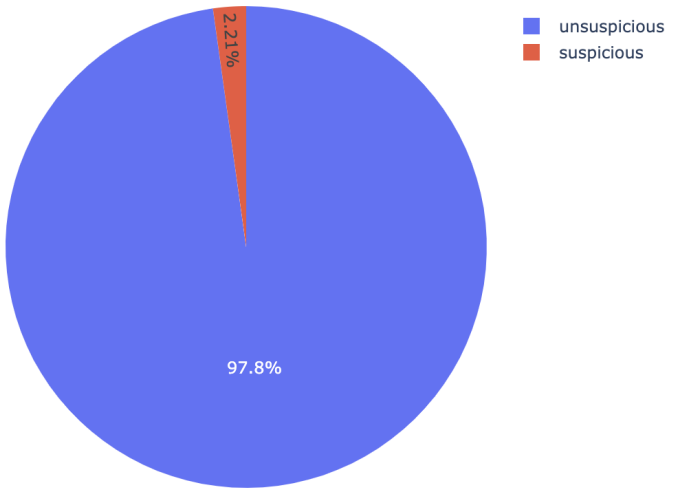

Figure 4: Suspicious UIDs proportion

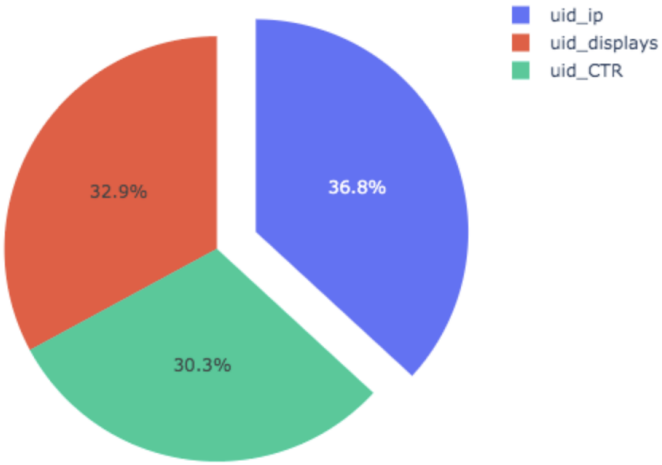

Figure 5: Suspicious UIDs repartition over patterns. We left the duplicate UIDs so that it is interpretable.

In the end, we see that all patterns have a comparable number of suspicious UIDs. Indeed, they range from 30.3% for the smallest to 36.8% for the biggest.

4 Re-implementation on Scala

Finally, 4 patterns were found when we did the offline analysis. We read the two topics *clicks* and *displays* that we stored in a Data Stream. We have defined variables as in offline, which allow us to set a threshold.

Once we had our data source, we had to deserialize the object so that the data was not in the form of string, but in the form of dictionaries. Thus, using the key *eventType*, we can know if this object corresponds to a click or a display.

Once deserialized, we have stored the rows as an **Event**. Event is a class that we created to read and access the elements more easily. Each attribute of this class corresponds to a key of a row in the DataStream.

To apply our rules over a time window, we relied on the notion of Watermark. For that, we used the *PunctuatedAssigner* class to associate a watermark to each timestamp.

We then implemented a **FraudDetector** object containing all the methods for fraudulent patterns detection, which we already implemented in offline in Python, but now in Scala. It is thus sufficient to create an object of this class and use its methods.

Finally, we wrote our outputs in text files that display the suspicious events.

5 Conclusion

To conclude, we identified 4 patterns where 3 of them concern potential fraudulent UIDs and the last one the potential fraudulent IPs. Our patterns turned out to be correct because we recovered a CTR closer to 10%.

Then, we can add that we kept 4 rules rather than 3, even if we could have the same uids between two rules for our specific time window, because we know that we could detect different UIDs from each rule.

Finally, we were challenged by the use of Scala but the biggest difficulty of the project was in fact the configurations where we could not initially read the data from Kafka. Through this project, we can notice the variety of suspicious behaviors that can be found on the web and how we can use Flink to read real-time stream.