TWEETOSCOPE REPORT

Ben Aissa Ahmed, Fellaji Mohammed, Mokbel Elie
December 1, 2020

1 INTRODUCTION

In this report, we go through the the architecture of the tweetoscope, and the main implementation steps and the choices we made in our approach to each aspect of the problem until we get to the final results. This report does not detail the deployment of the project, using Docker and Kubernetes, as this part will be showcased in the video presentation joined in the project.

2 ARCHITECTURE

In this section, we discuss the way we handled the implementation of the architecture of each node of the problem.

2.1 COLLECTOR

This is the main part in the development of the C++ code. The goal is to collect the different tweets generated by the generator and then send the cascades in two Kafka topics : cascade_series and cascade_properties. Since we are dealing with cascades, we started by creating a class **Cascade** for this object. We also added a class **Processor** that handles all the cascades having the same source id. Moreover, another class **ProcessorsHandler** was added which is responsible of managing all the processors. Whenever we receive a new tweet, we will use the operator "+" of the class ProcessorsHandler. This step will create the cascade if this is the first tweet of the cascade (thus of type "tweet") and create the processor in the case of a tweet with a new source id. Once created, cascades are stored in three tables. Because of that, shared and weak pointers are used. The queuing process is different from table to table and it depends on the topic used to send the kafka message : we either send a terminated cascade or

cascade after a given observation time. As a result, we added two attribute in the class Cascade : first_time and latest_time in which we keep track of the first and last times in a given cascade. In order to send the message using cppkafka, we found that it is better to only instantiate the producer once. This is why it was added as an attribute to the class ProcessorsHandler. Finally, we used the library spdlog to show the logs in the terminal. It is at the same time easy to use and fast compared to other libraries.

2.2 ESTIMATOR

This node is responsible for computing an estimation of the parameters (p, beta, G1) that characterise the Hawkes process by which we describe the retweets cascade (G1 will be useful later in the predictor).
We use a kafka consumer to read from "cascade_series" topic. For each cascade we compute the estimated parameters using a MAP estimator.
We can then make a first simple prediction using the Hawkes Process of the total size of the cascade.
The estimated parameters are then added to the "cascade_properties" messages and sent again to the topic using a kafka producer.

2.3 PREDICTOR

The predictor has a trickier process. It is mainly responsible for using the parameters of each cascade and a model sent by the learner node (explained in the next session) to make a prediction of the total size of each cascade. We use the following equation to make the prediction: $N_\infty = n + w_{Tobs}(\beta, n^*, G_1)\frac{G_1}{1-n^*}$ with $n^* = p\mu\frac{\alpha-1}{\alpha-2}$. w_{Tobs} is predicted by the model produced by the learner. Each T_obs has a different model.
The predictor also sends samples for the learner to train models to predict w_{Tobs}.

We want to describe here some of the main issues we encountered and how we addressed them.
First, the predictor needs a kafka consumer that reads messages from "cascade_properties". However, this topic contains two types of messages ('size' and 'parameters') and we need values from both for each cascade. We create create an empty dictionary before consuming the messages. This dictionary will have the cascade identifier as key and a dictionary with all the parameters we need as value. When we read messages we stock each parameter we need in this dictionary. Once we have read both type of messages for a given cascade(we check if all the parameters we need are in the dictionary) we can begin the computations.
We start by computing the true value of W_{obs} based on the parameters of the Hawkes process and the final size of the cascade. We send this value along these estimated parameters to the "samples" topic. They will be used by the learner for the training of W_{obs}.
We then extract the last model for the observation window of the current cascade. This part is actually done via a class model_consumer. In fact, we have created an object of this class for each observation window. The models are actually sent by the learner to a topic called "models", and each observation window has a corresponding partition in this topic. So, the

class model_consumer has the topic and partition as attributes as well as the observation window. We assign a consumer to each object model_consumer. For each cascade with this observation window, we check the last offset and extract the model in the message of that offset if it hasn't been read before.
That's how the model is refreshed each time and then used to output a prediction of W_{obs}. We compare this prediction with the real value and compute an ARE for performance and send it to a topic called "stats".

2.4 LEARNER

The learner reads messages in "samples", creates a dataset with the observation window, features and target W. Then each time we want to train new models, we split the dataset by observation window, and train Random Forest that will be sent to a specific partition in the topic "models". We train new models each 10 new samples.

3 CONCLUSION

Overall, we were able to overcome the many challenges we were faced with, and build a functioning application following the architecture detailed in the instructions. Moreover, the final error margin in our application is satisfying for a random forest problem, and allows us to make reliable predictions.