Rapport Tweetoscope

Mathieu Nalpon Mehdi Ech-Chouini Hamza Islah

December 2021

1 Présentation des principaux dossier

1.1 Hawkes_Processes

Dans ce dossier on retrouve une implémentation sous forme de classe du processus de Hawkes pour pouvoir l'utiliser de manière efficace dans différents fichiers (estimator.py, predictor.py).

1.2 Tweet_collection

Le dossier Tweet_collection regroupe tout ce qui concerne la génération de tweet et leur collection, les fichiers sont écrits en C++.

1.2.1 Tweet_generator (Fournit dans l'énonce)

Ce dossier contient les fichiers nécessaires pour la génération de tweets.

1.2.2 Tweet_collector

Ce dossier contient les fichiers nécessaires pour la collection de tweets. Comme décrit dans l'énoncé, deux classes ont été créées pour gérer les différents tweets en cascade : une classe Processor et une classe Cascade.

1.3 Pipeline_ai

Dans ce dossier, on retrouve la production en python de l'estimator, du predictor, et du learner.

1.3.1 predictor

Pour l'implantation du predictor V2, une forte attention a été portée à la façon dont on gère les divers types de messages qu'on reçoit dans les différents topics. Pour les valeurs provenant de cascade_series et de cascade_parameters, nous avons décidé d'utiliser des dictionnaires qui ont pour clé l'identifiant de la cascade et la fenêtre d'observation. Nous mettons à jour le modèle dans un dictionnaire selon la fenêtre d'observation. Lorsqu'il est possible de calculer une statistique, on vide les dictionnaires cascade_series et cascade_parameters en supprimant la clé, pour gérer la mémoire.

2 Déploiement

Notre solution tweetoscope est déployable sur kubernetes. Nous avons pu tester notre projet sur Minikube et Intercell. Nous avons produit deux fichiers pour chaque déploiement avec minikube-. . . et intercel-. . . .

2.1 Scalability

Notre solution est scalable pour le tweetcollecteur, estimateur, prédicteur et learner. Nous avons fait en sorte de mettre des partitions supplémentaires dans le déploiement des topics

En augmentant les partitions, lors de l'ajout de replicas, on peut diviser la tâche. Par exemple la collection des tweets en 4 pour optimiser le service. Nous n'avons pas eu de stratégie de partitionnement. Nous avons déterminé par

incrément et avons regardé la vitesse d'obtention d'information en fin de tâche (au niveau du learner). Démonstration disponible dans la vidéo.

2.2 Fault Tolérance

À la suite de notre déploiement sur Intercell. Nous avons effectué un test de fault tolérance. Nous avons pour cela supprimé un pod en marche sur le réseau. Nous observons bien la recréation d'un pod du service que nous avons supprimé. Démonstration dans la vidéo.

2.3 Docker Images and Git CI-CD

Notre projet est dockerisé. Nous avons produit des dockerfiles pour chaque services :

— Kafka : 204.52 Mi
— tweetGenerator : 337.37 MiB
— tweetCollector : 330.51 MiB
— Estimator : 225.08 MiB
— Predictor : 277.68 MiB
— Learner : 214.04 MiB
— Logger : 50.52 MiB

Le dossier Build regroupe tous les Dockerfiles

Notre pipeline est fonctionnelle. Elle effectue les build et push les images sur gitlab. Une modification d'un fichier par exemple pour le predicteur, entraînera un rebuild et push de l'image du prédicteur sur git. Démonstration dans la vidéo.

3 Perspectives d'amelioration

Nous aurions pu rendre le code écrit plus parlant en ajoutant par exemple des commentaires, des docstrings ou tenter de plus factoriser le code réalisé. Nous regrettons par manque de temps de ne pas pouvoir vous proposer cela. Nous n'avons qu'effleuré la génération de test et rapport de qualité, nous n'avons pas quelque chose de présentable pour ces deux rendus.