Rapport de PFE

Analyse de popularité des Tweets, déployée sur une architecture Apache Kafka

Auteurs :

Mohamed BEJAOUI
Quentin NAUDAN
Professeur : Frédéric PENNERATH

CentraleSupélec

Dernière modification : 31 mars 2020

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Contexte

Dans le cadre du nouveau cursus de CentraleSupélec, la majeure SIR sera remplacée dès la rentrée 2020 par la mention SDI-Metz d'avantage centrée sur les sciences des données et le machine learning. Dans ce contexte, le mini-projet robotique sera remplacé par un projet similaire dans l'esprit (i.e confrontation avec de nombreux outils et problèmes techniques liés au développement logiciel et à l'intégration dans une architecture logicielle complexe et distribuée) mais dont le thème sera centré sur les problèmatiques d'ingénierie des données sur le cloud.

Ce projet constituera la base de ce remplacement. Son objectif est de développer une méthode permettant la prédiction de popularité de tweets en temps réel. Pour la prédiction de popularité de tweets, on s'est basé sur un article scientifique [3] et pour assurer le traitement en temps réel, on a choisit d'intégrer notre solution dans une architecture Apache Kafka.

Pédagogiquement, ce projet doit être centré autour des cours d'apprentissage statistique, de machine learning et de big data de la mention SDI-Metz. Pour confronter le travail fait dans ce projet aux exigences du contexte, on a construit un cahier des charges montré ci-dessous.

Objectif	**Cours associé**	**Status / Commentaire**
Mise en œuvre d'une méthode d'apprentissage et son évaluation (cross val.)	App. aut.	Ok (random forest)
Mise en œuvre de concepts statistiques, modélisation probabiliste	Mod. stat.	Ok (processus de Hawkes, estimation MLE par optimisation)
Implémentation à partir d'un article scientifique		Ok
Richesse du problème et des données → possibilité de faire évoluer d'année en année les challenges		Ok (tweets, NLP)
Mise en œuvre d'un projet de développement logiciel (git, build, test)	Ing. appl. log.	
Intégration dans un framework logiciel avec une problématique de distribution du calcul et de passage à l'échelle	C++ Ing. appl. Log. Big Data ?	Ok (Kafka)
Implémentation d'un algorithme optimisé en C++. Compilation Cmake avec libs	C++	A faire (python non optimisé) Libs = Kafka, IPOPT
Déployer du code dans le Cloud (Docker/Kubernetes)	Ing. appl. log.	A faire. Contact chez Azure

FIGURE 1.1 – Cachier des charges du projet

1.2 Repository Gitlab

Le repository Gitlab est disponible à ce lien protoprojetsd9.git. Les principaux dossiers qui composent ce répertoire sont :

- *./* : Contient principalement les nodes Kafka sous fichiers python, ainsi que le script de configuration des topics kafka.
- *hawkes_point_process* : Contient les fonctions nécessaires pour l'optimisation de la log-vraisemblance, un notebook pour prendre la main sur chaque étape de l'optimisation et un exemple de jeu de données.
- *RF_model* : Contient un script qui mesure les performances du modèle, un script qui construit les paires (features, outputs) pour entrainer notre modèle ainsi que son CSV en sortie.
- *data* : les données du dataset NEWS utilisé dans l'article et nos données d'entrainement pour le modèle Random Forest.

```
protoprojetsd9
├── hawkes_point_process
│   ├── marked_hawkes.py
│   ├── marked_hawkes_point_process.ipynb
│   └── example_book.csv
├── RF_model
│   ├── performance_random_forest_model.py
│   ├── training_data_random_forest.py
│   └── training_data_RF.csv
├── data
│   ├── rf_train
│   │   └── training_data_rf.csv
│   └── tweets
│       ├── data.csv
│       └── index.csv
├── kafka_config.py
├── hawkes_params_fitter.py
├── random_forest_trainer.py
├── cascade_size_alert_receiver.py
├── tweets_data_reader.py
├── demo.sh
├── requirements.txt
└── README.md
```

Chapitre 2

Prédiction de popularité de Tweets

Cette première partie a pour objectif de mettre en place le modèle génératif d'une cascade de retweets, ainsi que la couche de prédiction supplémentaire.

2.1 Modèle génératif

2.1.1 La cascade de retweets

Nous extrayons comme données, pour un tweet publié, l'ensemble des retweets effectués à des temps t_i. Chaque utilisateur possède un nombre d'abonnés m_i (en ordonnée sur le graphique ci-dessous) que nous interprétons comme une magnitude d'influence de l'utilisateur ayant retweeté.

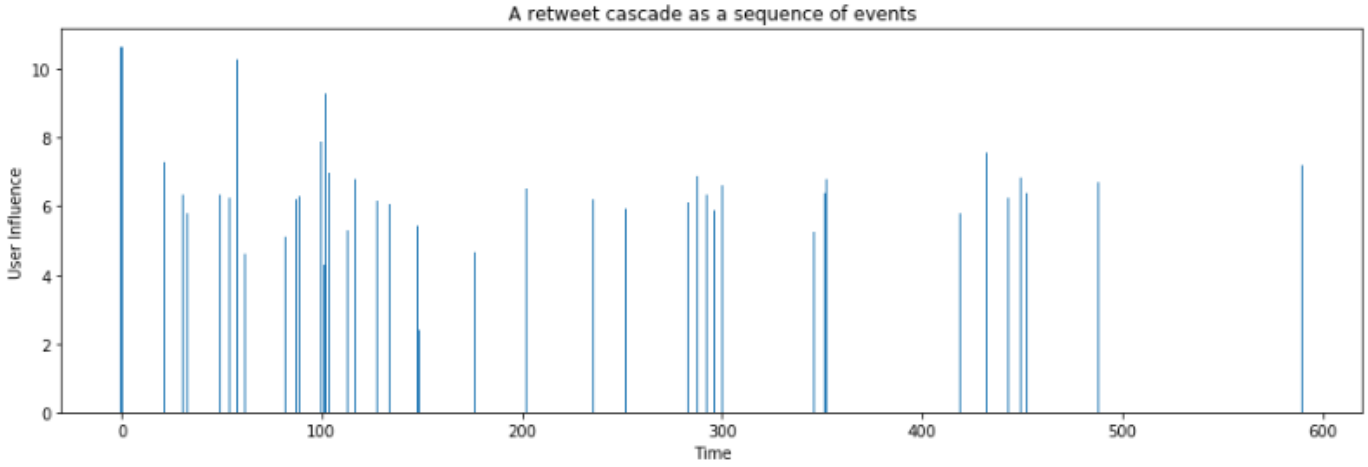

FIGURE 2.1 – Une cascade de retweets

Le nombre de pics est le nombre total de retweets. Chaque utilisateur ayant retweeté possède un nombre d'abonnés indiqué en ordonnée.

Nous souhaitons un **modèle génératif** pour décrire une cascade de retweets, afin d'en comprendre les méchaniques d'évolution. Nous utilisons, parmi les classes de processus stochastique, le *self-exciting point process* de Hawkes.

2.1.2 Hawkes self-exciting point process

Un processus de Hawkes se caractérise mathématiquement ainsi

$$\lambda(t) = \sum_{t_i<t} \nu(t - t_i)$$

où une fonction d'intensité conditionnelle (ici λ interprétée comme un taux d'arrivée de futurs retweets) est donnée par la somme des contributions (appelés noyaux) des événements passés aux temps antérieurs. Les événements passés rendent plus probable l'apparition de futurs événements.

Caractérisons le noyau d'un événement : sa contribution temporelle à générer de nouveaux retweets est modélisé par

$$\phi_{m_i}(t - t_i) = \kappa m_i^{\beta}(c + t - t_i)^{-(1+\theta)}$$

Le noyau d'un retweet fait intervenir 3 phénomènes :

- Un terme de qualité (ou viralité) inhérent au tweet d'origine, et partagé par tous les événements : κ
- Un terme d'influence du retweeteur sur ses abonnés : m_i^{β}, qui correspond à son nombre total d'abonnés déformé d'une puissance β.
- Un terme de mémoire dans le temps de la contribution du retweet. Pour charactériser l'inactivité de l'événement au bout d'un certain temps, ce terme décroit suivant une loi de puissance à la vitesse $1 + \theta$: $(c + t - t_i)^{-(1+\theta)}$. La constante c est un décalage temporel qui permet de ne pas faire diverger l'intensité du noyau au temps t proche de l'origine t_i.

Ci-dessous les graphes de deux noyaux, aux memes paramètres $(\kappa, \beta, c, \theta)$, pour des couples (temps, magnitude) différents.

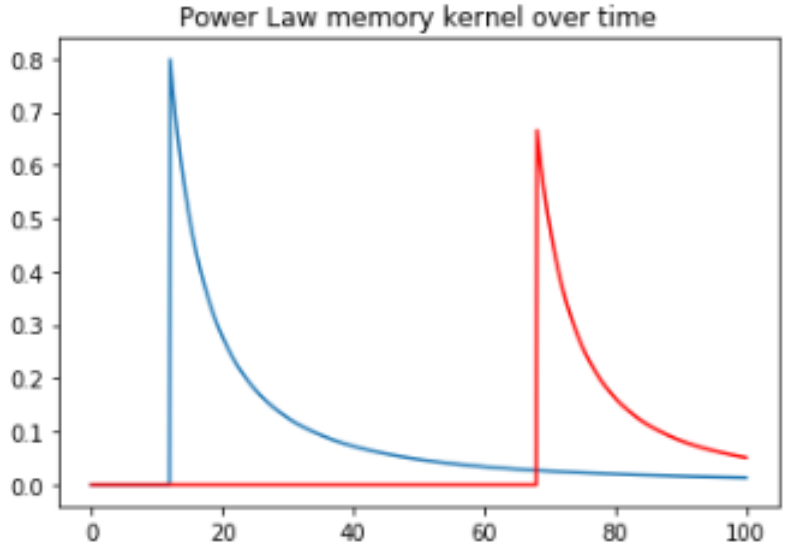

FIGURE 2.2 – Graphes de deux noyaux aux couples (temps, magnitude) différents

Les paramètres sont $\kappa = 0.8, \beta = 0.6, c = 10, \theta = 0.8$
En bleu : retweet de noyau $\phi_{1000}(t - 12)$; en rouge : retweet de noyau $\phi_{750}(t - 68)$

Pour résumer, les contributions des événements donnent la fonction de taux d'arrivée de nouveaux retweets

$$\lambda(t) = \sum_{t_i<t} \phi_{m_i}(t - t_i)$$

Pour une même cascade de retweets (même tweet d'origine et mêmes retweets caractérisés par leurs couples (temps, magnitude)), nous traçons ci-dessous deux fonctions λ pour des paramètres $(\kappa, \beta, c, \theta)$ différents.

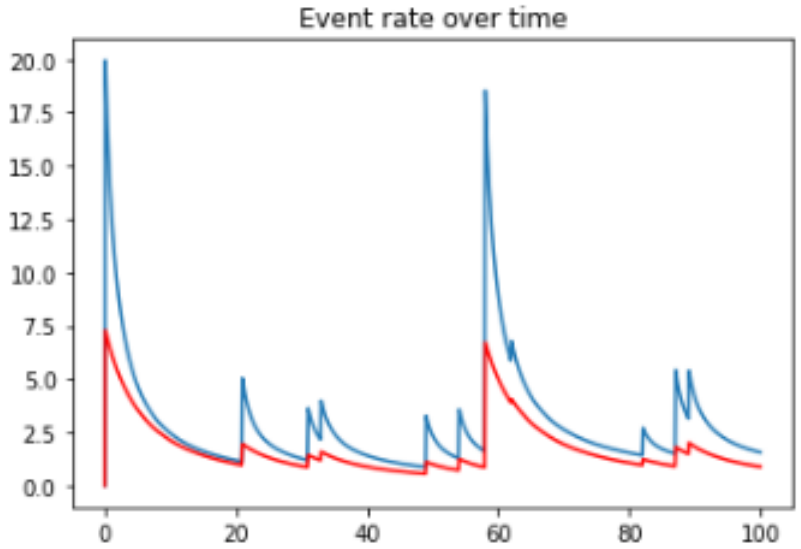

FIGURE 2.3 – Graphes de λ pour des tuples de paramètres différents

En bleu : $\kappa = 0.24, \beta = 0.5, c = 2, \theta = 0.2$;
En rouge : $\kappa = 0.8, \beta = 0.6, c = 10, \theta = 0.8$

A ce stade nous ne savons pas évaluer la fonction λ sans connaitre les 4 paramètres $(\kappa, \beta, c, \theta)$. Nous les déterminons par approche bayésienne : nous confrontons notre modèle aux données observées sur un temps d'observation T, en évaluant la **log-vraisemblance**

$$L(\kappa, \beta, c, \theta) = \sum_{i=1}^{n} log(\lambda(t_i)) - \int_{t=0}^{T} \lambda(\tau)\ \mathrm{d}\tau$$

Cette log-vraisemblance nous est donnée par la bibliographie sur les *point process* de Hawkes. Nous cherchons un maximum local de cette fonction au moyen d'une librairie open source : **Ipopt** (Interior Point OPTimizer) [1].

2.1.3 Optimization non-linéaire

Nous importons la librairie dans notre environnement Anaconda. La classe du solver est construite dans le fichier *hawkes_point_process/marked_hawkes.py*. Premièrement, le problème est identique à minimiser l'opposé de la log-vraisemblance, qui s'évalue grace aux données ainsi :

$$-L(\kappa, \beta, c, \theta) = \kappa \sum_{i=1}^{n} m_i^{\beta} \left[\frac{1}{\theta c^{\theta}} - \frac{1}{\theta (c + T - t_i)^{\theta}} \right] - \sum_{i=2}^{n} log(\kappa) - \sum_{i=2}^{n} log \left(\sum_{t_j<t_i} \frac{m_j^{\beta}}{(c + t_i - t_j)^{1+\theta}} \right)$$

Remarquons que nous avons une expression analytique de l'intégrande de λ. Ensuite, nous exprimons les 4 dérivées partielles pour évaluer la jacobienne de la log-vraisemblance négative :

$$-\frac{\partial L}{\partial \kappa} = \sum_{i=1}^{n} m_i^{\beta} \left[\frac{1}{\theta c^{\theta}} - \frac{1}{\theta (c + T - t_i)^{\theta}} \right] - \frac{n-1}{\kappa}$$

$$-\frac{\partial L}{\partial \beta} = \kappa \sum_{i=1}^{n} m_i^{\beta} \; log(m_i) \left[\frac{1}{\theta c^{\theta}} - \frac{1}{\theta (c + T - t_i)^{\theta}} \right] - \sum_{i=2}^{n} \left(\frac{\sum_{t_j < t_i} \frac{m_j^{\beta} \; log(m_j)}{(c + t_i - t_j)^{1+\theta}}}{\sum_{t_j < t_i} \frac{m_j^{\beta}}{(c + t_i - t_j)^{1+\theta}}} \right)$$

$$-\frac{\partial L}{\partial c} = \kappa \sum_{i=1}^{n} m_i^{\beta} \left[\frac{1}{(c + T - t_i)^{1+\theta}} - \frac{1}{c^{1+\theta}} \right] - \sum_{i=2}^{n} \left(\frac{\sum_{t_j < t_i} \frac{-(1+\theta) \; m_j^{\beta}}{(c + t_i - t_j)^{2+\theta}}}{\sum_{t_j < t_i} \frac{m_j^{\beta}}{(c + t_i - t_j)^{1+\theta}}} \right)$$

$$-\frac{\partial L}{\partial \theta} = \kappa \sum_{i=1}^{n} \frac{m_i^{\beta}}{\theta^2} \left[\frac{1 + \theta log(c + T - t_i)}{(c + T - t_i)^{\theta}} - \frac{1 + \theta log(c)}{c^{\theta}} \right] - \sum_{i=2}^{n} \left(\frac{\sum_{t_j < t_i} \frac{-log(c + t_i - t_j) m_j^{\beta}}{(c + t_i - t_j)^{1+\theta}}}{\sum_{t_j < t_i} \frac{m_j^{\beta}}{(c + t_i - t_j)^{1+\theta}}} \right)$$

L'algorithme nécessite un point initial pour procéder à l'optimisation itérative. Ce point initial est choisi aléatoirement en respectant les intervalles d'existences des variables à optimiser :

$$\begin{cases} \kappa > 0 \\ 0 < \beta < 1.016 \text{ (voir paragraphe suivant)} \\ c > 0 \\ \theta > 0 \end{cases}$$

De plus, la contrainte non-linéaire suivante s'applique sur les variables (son existence est expliquée au paragraphe suivant) :

$$\begin{cases} g \; : \; (\kappa, \beta, c, \theta) \rightarrow log\Big(\kappa \frac{\alpha - 1}{\alpha - \beta - 1} \frac{1}{\theta c^{\theta}}\Big) < 0 \\ \text{de Jacobienne } J_g \; (\kappa, \beta, c, \theta) = \Big[\frac{1}{\kappa}, \; \frac{1}{\alpha - \beta - 1}, \; -\frac{\theta}{c}, \; -\frac{1}{\theta} - log(c)\Big] \end{cases}$$

L'exemple ci-dessous permet de tester notre algorithme d'optimisation sur des cascades de retweets tirées de la base de données de l'article.

```
cd hawkes_point_process/
conda activate <MY_ENV>
python test_ipopt.py
```

2.1.4 Prédiction du nombre de retweets

Ayant désormais la connaissance de notre intensité temporelle λ (en optimisant ses 4 paramètres), nous pouvons prédire le nombres de retweets générés au cours du temps. L'une des grandeurs clés qui décrivent un processus de Hawkes est le **branching factor**, c'est le nombre d'enfants attendus générés à partir d'un seul événement. L'intensité conditionnelle d'un seul événement se réduit à son noyau $\phi_m(\tau)$. Pour en déduire un nombre d'événements futurs engendrés, nous intégrons sur le temps mais aussi sur la magnitude du retweet. La magnitude m est la réalisation d'une variable aléatoire, en effet tous les utilisateurs de Twitter suivent une distribution en loi de puissance sur leur nombre d'abonnés m. Cette loi $P(m) = (\alpha - 1)\, m^{-\alpha}$ caractérise bien la faible proportion d'utilisateurs ayant un grand nombre d'abonnés contre les utilisateurs ayant très peu d'abonnés, son paramètre a été estimé à $\alpha = 2.016$.

Le branching factor se calcule donc par

$$n^* = \int_{m=1}^{\infty} \int_{\tau=0}^{\infty} P(m)\phi_m(\tau)\ \mathrm{d}\tau \mathrm{d}m = \kappa \frac{\alpha - 1}{(\alpha - \beta - 1)\theta c^{\theta}}$$

On en déduit les conditions sur les variables (introduites précédemment)

$$\theta > 0 \text{ et } 0 < \beta < \alpha - 1.$$

Nous estimons qu'un événement génère n^* enfants, cette génération va à son tour faire apparaitre $(n^*)^2$, et ainsi de suite. D'où le nombre total de retweets générés est la somme géométrique $\sum_{k=1}^{\infty}(n^*)^k$. Par exemple, pour un branching factor $n^* = \dfrac{2}{3}$, alors 2 retweets générés sont attendus.

Lorsque $n^* \geq 1$, nous sommes en régime super-critique et notre modèle génératif ne peut pas prédire (car le nombre de rewteets générés attendus est infini), ceci apporte une **contrainte d'optimisation** sur les variables, la fonction g introduite précédemment.

Après avoir étudié un seul événement, nous considérons tous les retweets de la cascade. Nous notons A_1 le nombre d'événements enfantés par la 1ère génération de retweets observés (en rouge sur le graphe ci-dessous).

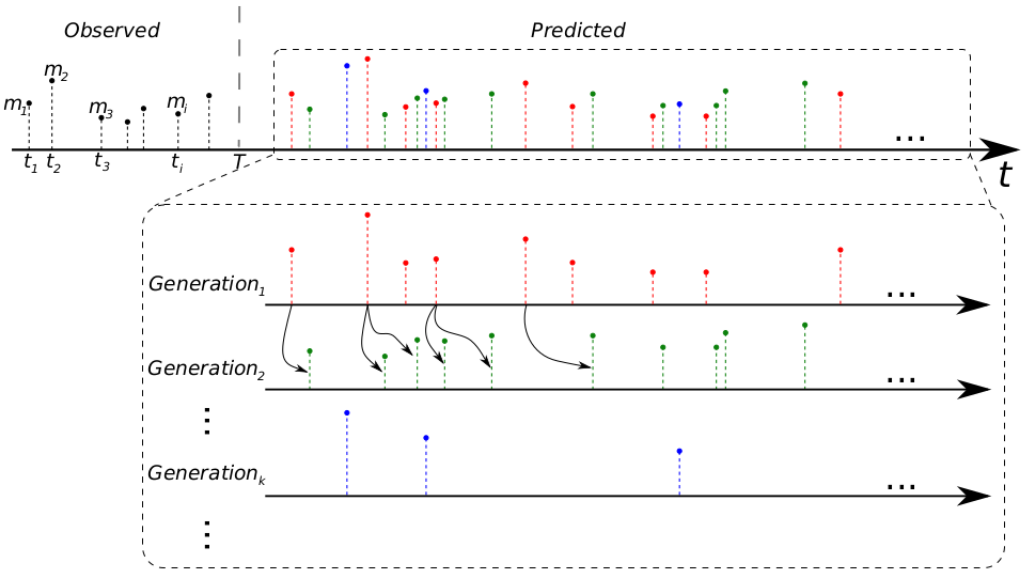

FIGURE 2.4 – Générations successives de retweets à partir de notre cascade observée

On calcule aisément A_1 par l'intensité conditionnelle intégrée sur tout le temps à partir du dernier retweet observé :

$$A_1 = \int_T^\infty \lambda(t)\ \mathrm{d}t$$

Ensuite, nous estimons chaque génération par la récurrence $A_{i+1} = A_i * n^*$, une génération engendre en moyenne son branching factor. D'où la prédiction finale du total de retweet s'obtient, quand $n^* < 1$, par :

$$N_\infty = n + \sum_{i=1}^{\infty} A_i = n + \frac{A_1}{n^* - 1}$$

Cette estimation est en réalité très peu performante, la prédiction peut souvent être, en ordre de grandeur, 10 fois la valeur réelle. Un modèle génératif est efficace pour décrire les méchanismes, mais peu performant pour prédire. Ainsi, nous ajoutons une couche de prédiction supplémentaire, détaillée à la section suivante.

2.2 Couche de prédiction supplémentaire

2.2.1 Objectif et choix du modèle

Nous avons effectué des hypothèses lourdes pour notre modèle génératif qui ne reflètent pas la réalité :

— Le temps de vie d'une cascade de retweets peut etre inhomogène : les premiers retweets ont une contribution d'une durée de vie courte tandis que les derniers retweets allongent le temps de mémoire. Ceci n'est pas capturé par le terme de mémoire en loi de puissance.

— Le nombre d'utilisateurs atteints par un retweet et ayant une propension à retweeter n'est pas capturé par le terme d'influence (le nombre d'abonnés d'un déformé d'une puissance β commune à tous les retweets).

Nous souhaitons réaliser l'apprentissage supervisé d'un **scaling factor w** pour ajuster la prédiction finale de retweets :

$$\widehat{N}_\infty = n + w\frac{A_1}{n^* - 1}$$

Notre choix de features se porte sur les grandeurs introduites dans notre modèle génératif : $\kappa, \beta, c, \theta, n^*, A1$. Le paramètre κ est redondant car linéairement proportionnel aux grandeurs n^* et $A1$, et le paramètre β a été découvert non correlé à l'erreur de prédiction donc inutile comme feature du modèle.

Une très faible variation de nos features (c, θ) entraine une forte variation des grandeurs $(n^*, A1)$ et du scaling factor w. Pour palier à cette non-linéarité, nous préférons utiliser une Random Forest qui prend en entrée les features $\{c, \theta, A1, n^*\}$. Nous n'avons pas tuner la profondeur des arbres de décisions, cette étape est nécessaire pour majorer la taille mémoire du modèle qui sera propagé aux serveurs via un topic kafka.

2.2.2 Performances

Notre algorithme appartient à la classe des **régressions**, et notre cout d'erreur doit etre comparable pour différents ordres de grandeur (dans le cadre de notre problème : mal estimer 140 retweets au lieu de 100 est similaire à mal estimer 1400 retweets au lieu de 1000). Pour mesurer la performance de la régression, nous utilisons alors **l'absolute relative error** :

$$ARE = \frac{\|N_{real} - \widehat{N}_{\infty}\|}{N_{real}}$$

Les auteurs de l'article obtiennent, avec un temps d'observation de 10 minutes, un ARE moyen égal à 0.20, c'est-à-dire que le nombre total de retweets a un écart moyen de 20% du total réel.

Nous présentons nos résultats sous forme de **boxplots**, qui exhibent les ARE médian (ligne jaune), moyen (pointillés rouges) et quartiles. 10 boxplots sont obtenus en regroupant les données par déciles de popularité (les déciles sont des intervalles sur la variable N_{real} qui contiennent chacun 10% des cascades).

Ci-dessous notre résultat de performance, l'ARE total moyen est égal à 34.95% :

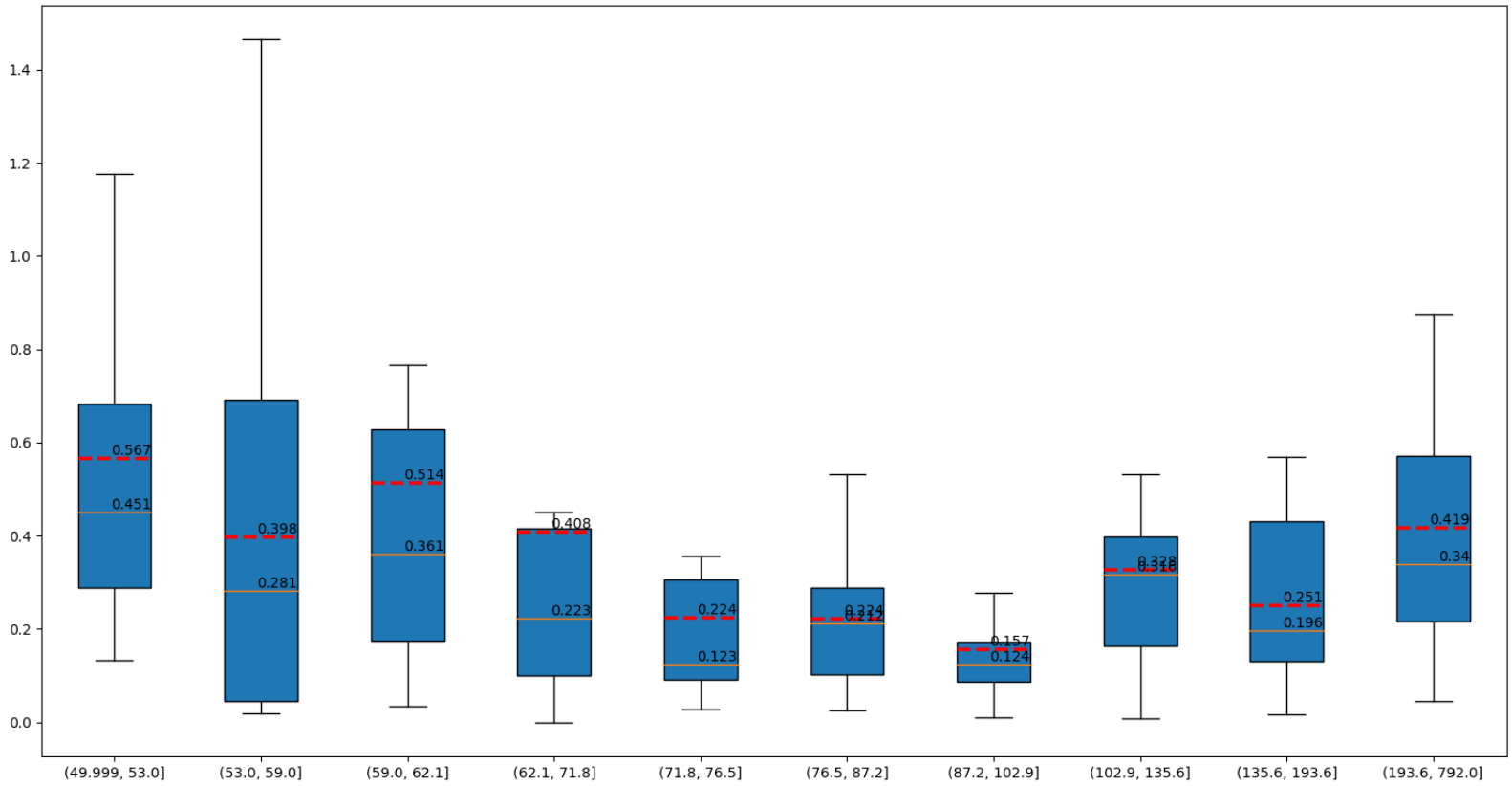

FIGURE 2.5 – ARE Total Moyen = 34.95% pour 168 données testées et 1512 données d'entrainement

Nous n'avons pas limité la taille de notre modèle (profondeur des arbres, nombre d feuilles maximal, ect), ce qui nécessite de porte attention à la taille du message transmis sur un topic kafka (limité à $1Mo$). De fait nous limitons le nombre de données d'entrainement à 500, ci-dessous notre performance sous cette contrainte :

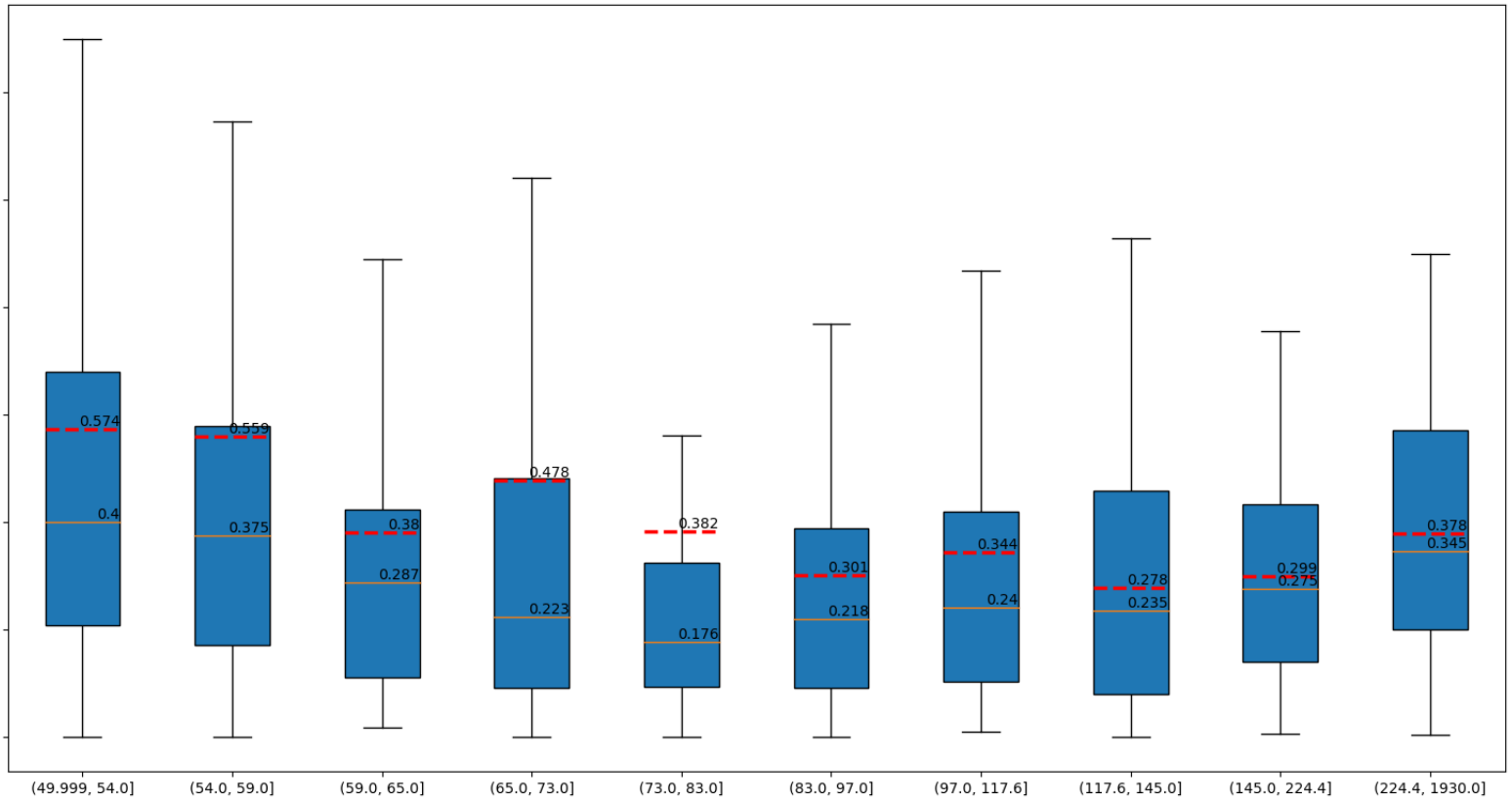

FIGURE 2.6 – ARE Total Moyen = 39.8% pour 1029 données testées et 499 données d'entrainement

Chapitre 3

Architecture Kafka de notre projet

Le but de cette partie est d'introduire Apache Kafka [2] et de présenter la manière dont on y a intégré notre solution.

3.1 Notions de Kafka

Apache Kafka est une plateforme distribuée de diffusion de données en continu gérée par la fondation Apache. Elle est capable de publier, stocker, traiter et souscrire à des flux d'enregistrement en temps réel.

Étant rapide, scalable, durable et tolérant aux défaillances, Kafka propose des fonctionnalités très utiles à notre solution dans le cadre d'estimation de popularité des tweets en temps réel.

On va commencer par présenter quelques notions de bases importantes de Kafka qui sont au coeur de notre architecture. Avant cela, voici un schéma décrivant le fonctionnment général de Kafka :

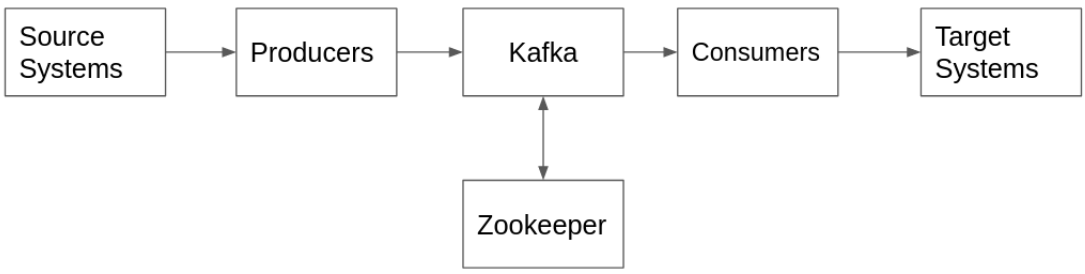

FIGURE 3.1 – Fonctionnement de Apache Kafka

En gros, les données reçues par Apache Kafka sont conservées au sein de topics. Chaque topic correspond à une catégorie de données. Les systèmes qui publient des données dans les topics Kafka sont des Producers. Les systèmes qui lisent les données des topics sont quant à eux des Consumers.

3.1.1 Topics

Le **topic** est un flux de données particulier, uniquement identifiable par son nom. Chaque topic parallélise son flux de messages sur plusieurs **partitions**. Une partition est un objet ordonné (par ordre d'arrivée) et immuable (donnée écrite non modifiable). Les enregistrements dans les partitions reçoivent chacun un numéro d'identification séquentiel appelé **offset** qui identifie de manière unique chaque enregistrement dans la partition.

Nos messages transmis sont généralement une paire (clé, donnée) dont la clé permet l'assignation à une partition (sans celà l'assignation est aléatoire et équilibrée).

Ci-dessous un schéma du flux des messages, identifiés par des offsets, au sein des partitions d'un topic Kafka.

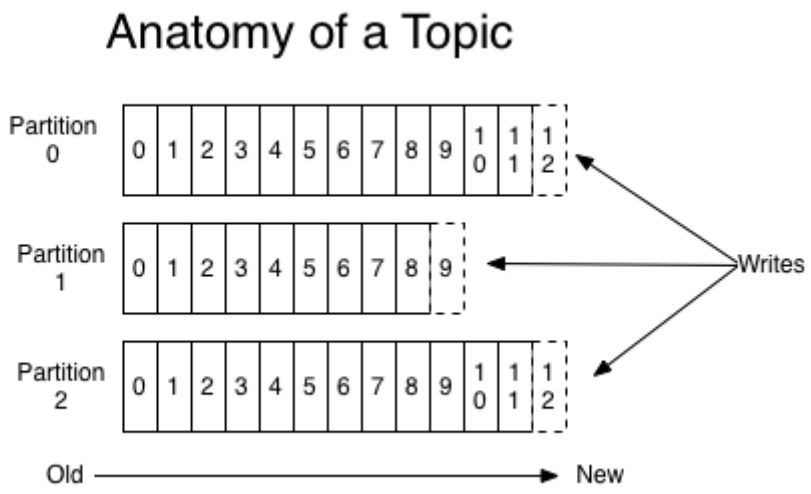

FIGURE 3.2 – Flux de messages via les partitions et lecture des offsets

3.1.2 Producers

Le producer publi les données sur les topics de leur choix. Il est responsable du choix de l'affectation des messages publiés à telle ou telle partition dans un topic. Cela peut se faire de manière circulaire simplement pour équilibrer la charge ou selon une fonction de partition sémantique (par exemple, en fonction d'une clé de message).

3.1.3 Consumers

Le consumer lit les messages depuis les topics, les messages provenant de la même partition étant lus dans l'ordre.

Les consumers s'attribuent un nom de **consumer group**, et chaque message publié à un topic est remis à une instance de consumer au sein de chaque consumer group abonné. Les instances de consumers peuvent se trouver dans des processus séparés ou sur des machines séparées.

Si toutes les instances de consumers ont le même consumer group, alors les messages seront effectivement répartis entre les instances de consumers de manière équilibrée.

Si toutes les instances de consumers ont des consumer groups différents, alors chaque message sera diffusé à tous les processus de consumers.

Voici un schéma résumant ceci :

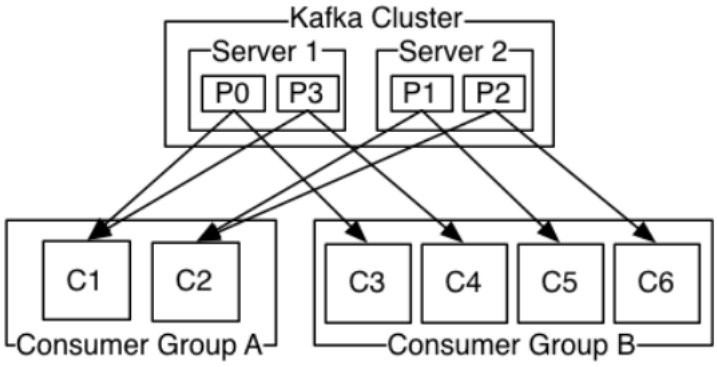

FIGURE 3.3 – Un cluster Kafka à deux serveurs hébergeant quatre partitions (P0-P3) avec deux consumer groups

Le consumer group A a deux instances de consumers et le groupe B en a quatre.

Pour plus d'information concernant les notions et fonctionnalités de Kafka, consultez la documentation officielle.

3.2 Description de notre architecture

On va commencer par présenter un schéma génaral de l'architecture complète de notre solution. Cette architecture sera détaillée dans la suite de cette partie.

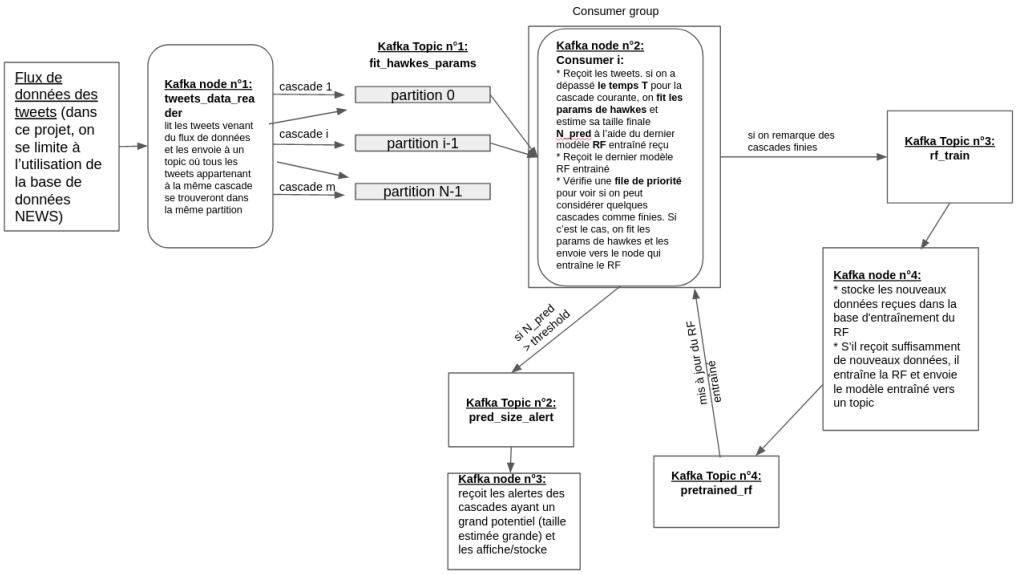

FIGURE 3.4 – Vue d'ensemble de notre architecture

3.2.1 Topics kafka

Décrivons les topics de notre architecture :

— *fit_hawkes_params* gère le flux d'information sur les retweets. Chaque message prend la forme de la paire (temps, magnitude) propre au retweet, et la clé du message est donnée par le numéro de cascade à laquelle le retweet appartient. Tous les retweets d'une même cascade sont, avec certitude, regroupés dans la même partition. Dans ce topic, le flux est parallélisé sur 100 partitions.

— *rf_train* gère le flux de données d'entrainement pour l'algorithme d'apprentissage supervisé, la donnée d'un message est le tuple $(c, \theta, n^*, A1, w)$. Lorsqu'une cascade de retweets atteint sa fin de vie (il n'y aura vraisemblablement plus de futur retweet), on calcule les features à partir de la cascade tronquée jusqu'au temps d'observation fixé et on calcule l'output (le scaling factor w) à partir de la taille finale de la cascade. Il est inutile de donner une clé au message, les données peuvent etre assignées sur une partition quelconque des 100 partitions disponibles.

— *pretrained_rf* gère le flux du dernier modèle mis à jour de notre Random Forest, ce topic possède 1 partition et le message est notre modèle sous format 'bytes' compressé après sérialisation pickle.

— *pred_size_alert* gère le flux des alertes au sujet des cascades dont la taille estimée dépasse un seuil fixé. La donnée du message contient le numéro de la cascade et sa taille estimé. Il n'est pas nécessaire d'ajouter une clé. Ce topic possède 20 partitions.

Nos topics sont créés lors de l'éxécution du script de configuration de kafka comme ci-dessous :

```
conda activate <MY_ENV>
python kafka_config.py
```

3.2.2 Nodes kafka

Un **node** n'est pas une terminologie liée à Kafka mais plutôt une qu'on a adoptée lors de notre projet. En fait, ce qu'on appelle un node est un script qui joue le rôle de **producer(s)** ou de **consumer(s)** ou les deux en même temps. Les **nodes** kafka sont les fichiers python dans le dossier principal de notre repository Git. Ci-dessous le schéma synthétique d'organisation de nos nodes et topics :

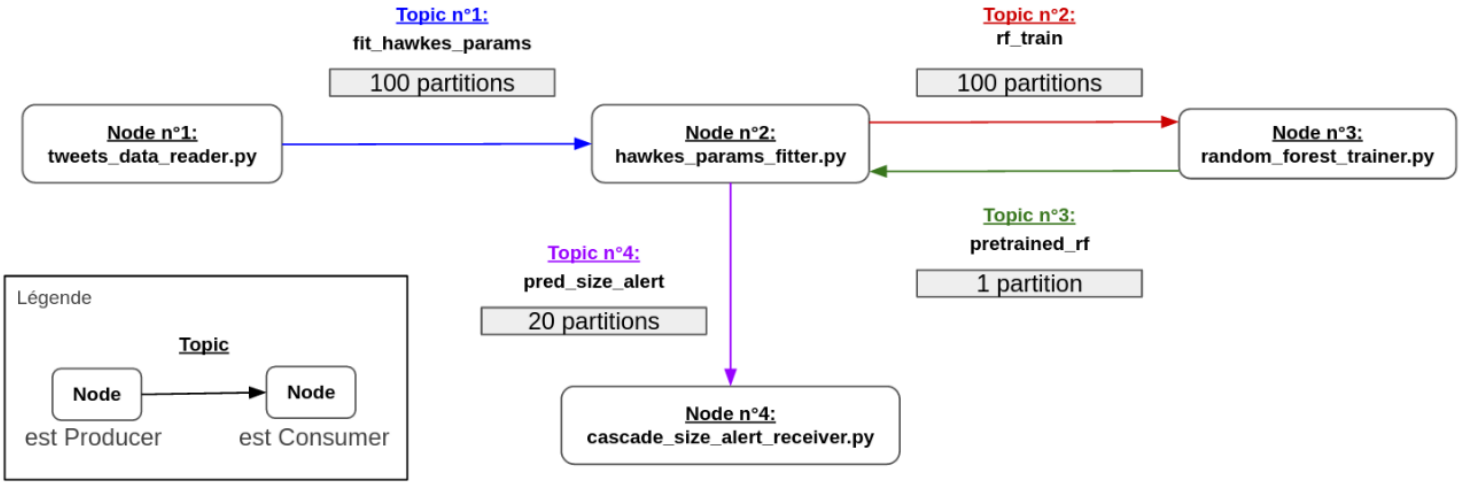

FIGURE 3.5 – Schéma d'organisation des nodes et topics kafka

Décrivons en détail le rôle de nos nodes :

— *tweets_data_reader.py* émet les données des retweets depuis la base de données NEWS (sous fichiers csv) vers le topic *fit_hawkes_params*. Nous émettons aléatoirement des retweets, tout en respectant l'ordre temporel au sein d'une même cascade. Aussi, pour rendre la simulation du flux de données des tweets plus réaliste, on rajoute, entre l'envoie de deux tweets successifs, un délai aléatoire de l'ordre du centième de seconde qui suit une loi normale.

— *hawkes_params_fitter.py* est le node central de l'architecture, c'est un node que nous souhaiterions déployer sur plusieurs **brokers** (serveurs, voir section suivante) pour un fonctionnement distribué et donc plus rapide. Ce node exécute, en parallèle, 3 **threads** pour les 3 objectifs suivants :

  — Le 1^{er} thread est consumer du topic *pretrained_rf* où il acquière le dernier modèle mis à jour de Random Forest. Avant qu'un premier modèle entrainé soit reçu sur ce node, le scaling factor propre à la couche de prédiction est intialisé à $w = 1$ (ce qui correspond à la prédiction issue du modèle génératif, sans couche supplémentaire de prédiction).

  — Le 2^{nd} thread se place en consumer du topic *fit_hawkes_params* pour lire les données des retweets, en regroupant les tweets appartenant à la même cascade. A la réception d'un retweet, si le temps d'observation est dépassé, alors il entame la phase de prédiction. Pour rappel, cette prédiction consiste à trouver le modèle génératif le plus vraisemblable (par optimisation de la log-vraisemblance de la fonction λ : le taux d'arrivée d'événements) et appliquer une couche de prédiction (algorithme de Random Forest). Si le total de retweets prédit dépasse une valeur seuil, alors cette information est transmise vers le topic *pred_size_alert*, auquel ce thread se positionne en tant que producer.

— Le 3^{me} thread évalue, à partir des données reçues du topic *fit_ hawkes_ params*, quelle cascade de retweets peut etre estimée comme terminée (aucun retweet futur n'est potentiellement attendu). Notre critère pour considérer une cascade comme finie est lorsqu'elle ne reçoit plus de retweets depuis une durée supérieure à une valeur seuil, alors elle est utilisée comme donnée d'entrainement à la Random Forest. C'est-à-dire que ses paramètres $\{c, \theta, A1, n^*\}$ sont évalués sur les retweets contenus dans le temps d'observation, et le scaling factor w est calculé à partir du nombre total de retweets constaté (supposé comme le total réel). Ces données sont transmises vers le topic *rf_ train*, auquel le thread se positionne en tant que producer.

— *random_ forest_ trainer.py* est le node qui entraine le modèle de Random Forest. Il est consumer du topic *rf_ train* (données d'entrainement issues des cascades de retweets terminées) et re-entraîne le modèle dès qu'il reçoit un certain nombre de nouveaux cascades. Ensuite, le modèle est compressé, à l'aide de pickle, sous format 'bytes' et transmis au topic *pretrained_ rf*, auquel ce node se place en tant que producer.

— *cascade_ size_ alert_ receiver.py* affiche les cascades de retweets qui méritent notre attention (dont le total de retweets prédit dépasse une valeur seuil fixée). Il est consumer du topic *pred_ size_ alert*.

3.2.3 Brokers kafka

Un **broker** kafka est modélisé comme un serveur qui héberge des topics. En mettant en place plusieurs brokers, on augmente considérablement la capacité de parallélisation du traitement des tweets car ceci nous offre la possibilité de distribuer ce traitement sur plusieurs machines.

Faute de temps, on n'a, malheureusement, pas pu expérimenter avec les brokers mais d'après nos recherches, l'adaptation de notre code existant à plusieurs brokers doit se faire aisément sans présenter une grande difficulté technique.

3.3 Démonstration globale

Cette section met en application l'utilisation de notre architecture kafka : activation des serveurs ZooKeeper et Kafka, configuration des topics, et éxécutions des nodes. L'ensemble des commandes à écrire dans le terminal sont gérés par le script bash ***demo.sh***.

3.3.1 demo.sh

Dans le terminal de notre repository Git, éxécutez :

```
# Renseigner avant le nom d'environnement Anaconda et les bons /path/to/folder
./demo.sh
```

Ce script bash ouvre un terminal et 7 onglets (il est nécessaire de configurer les onglets pour qu'ils ne se ferment pas après éxécution : "Edit" → "Preferences" → "Command" → "When command exits : Hold the terminal open").

Décrivons l'action de chaque onglet :

— L'onglet *"zookeeper"* initialise le serveur Zookeeper dont kafka est dépendant, il éxécute la commande :

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

— Après un court temps d'attente (3 secondes), l'onglet *"kafka"* lance le serveur Kafka, il éxécute la commande :

```
bin/kafka-server-start.sh config/server.properties
```

Le serveur parcourt les topics d'une éxécution précédente, et enregistre la position des offsets de chaque patition. Il est nécessaire de patienter 15 secondes avant de créer nos topics et éxécuter nos nodes.

— Après un long temps d'attente (15 secondes de sécurité), l'onglet *"config"* crée nos topics si ce n'est pas déjà fait, il éxécute la commande :

```
conda activate <MY_ENV>
python kafka_config.py
```

Il est possible que l'éxécution échoue et annonce aucun broker disponible, il faut alors purger les topics (voir section suivante).

— Après un court temps d'attente (3 secondes), l'onglet *"rf_trainer"* lance le node *random_forest_trainer.py*, il éxécute la commande :

```
conda activate <MY_ENV>
python random_forest_trainer.py
```

— L'onglet *"alert_receiver"* lance le node *cascade_size_alert_receiver.py*, il éxécute la commande :

```
conda activate <MY_ENV>
python cascade_size_alert_receiver.py
```

Sur cet onglet vont s'afficher les prédictions de nombre total de retweets dépassant une valeur seuil (de plus les données sont transmises à ce node seulement si le modèle de Random Forest a été entrainé au moins une fois, ce qui se traduit par la variable globale du *scaling factor* w qui n'est plus égale à 1).

— L'onglet *"tweet_process"* lance le node *hawkes_params_fitter.py*, il éxécute la commande :

```
conda activate <MY_ENV>
python hawkes_params_fitter.py
```

Sur cet onglet vont s'afficher les messages de cascades en cours de traitement, du nombre d'itération pour optimiser les paramètres de Hawkes, et si la cascade est en régime super-critique ($n^* \geq 1$)

— L'onglet *"tweet_reader"* lance le node *tweets_data_reader.py*, il éxécute la commande :

```
conda activate <MY_ENV>
python tweets_data_reader.py
```

Ce node est éxécuté un court délai (3 secondes) après les autres nodes.

3.3.2 Commandes de routine

Dans le cas où on recontre des problème d'éxécution dans certain nodes, en particulier *hawkes_params_fitter.py*, il est parfois nécessaire de nettoyer certains topics pour éliminer les consumers offsets et recommencer la lecture des données de zéros. Pour faire ceci, il faut lancer les deux commandes suivantes pour le topic cible (n'oubliez pas de changer le nom du topic dans les commandes). Il est indisponsable d'attendre au moins une minute avant l'éxécution de la deuxième commande.

```
bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic mytopic \
--config retention.ms=1000

# ... wait a minute ...

bin/kafka-topics.sh --zookeeper localhost:2181 --alter --topic mytopic \
--delete-config retention.ms
```

Chapitre 4

Conclusion & pour aller plus loin

4.1 Conclusion

Dans ce PFE, nous avons implémenter une solution issue d'un article scientifique avec une performance raisonnablement proche de celle obtenue par les auteurs. Le sujet fait intervenir un modèle statistique (*self-exciting point process* de Hawkes) et un modèle d'apprentissage supervisé (Random Forest Regressor).

Nous avons intégrer notre solution dans une architecture Apache Kafka, pour prouver la possibilité d'un traitement distribué, en temps réel et robuste aux défaillances. Il nous manque la composante de scalabilité par la distribution de nos nodes et partitions sur plusieurs serveurs.

4.2 Pour aller plus loin

Nous résumons les actions que nous aurions aimé mettre en place :

— Créer des brokers et *consumer groups* pour distribuer sur plusieurs machines l'éxécution de nos noeuds.
— Suite à la parrallélisation accrue, surveiller la tolérance aux défaillances de nos topics (via leur *replication factor* par exemple).
— Obtenir en temps réel les retweets, via l'API gratuite de Twitter, pour les tweets que nous souhaitons suivre.
— Améliorer la performance de l'estimation de popularité de tweets en combinant, avec la solution actuelle, une approche basée exclusivement sur des features pouvant être extraites via l'API Twitter. Cette deuxième approche est expliquée dans l'article [3].
— Déployer notre solution sur le cloud avec **Docker**/**Kubernetes**.

Bibliographie

[1] Création d'une classe de solver avec Ipopt, exemple à ce lien : pythonhosted.org/ipopt.

[2] Documentation et *Quickstart* d'Apache Kafka à ce lien : kafka.apache.org/quickstart.

[3] Swapnil Mishra, Marian-Andrei Rizoiu et Lexing Xie. "Feature driven and point process approaches for popularity prediction". In : *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 2016, p. 1069-1078.