

Michon Raphaël

2 juin 2024 – 2 août 2024

Rapport de stage 2A

LORIA



Sujet : Analyse et annotation des logs via des algorithmes d'apprentissage automatique

Tuteur : M. Lahmadi

Filière : ISN, promotion 2025



## Table des matières

I.	Introduction .....	4
II.	État de l'art .....	5
1.	Templatisation (log parsing) .....	5
2.	Détection d'anomalie.....	9
a.	Non-supervisé .....	9
b.	Semi-supervisé.....	17
c.	Supervisé .....	19
III.	Résultats .....	25
1.	Configuration machine .....	25
2.	Méthode de test .....	25
IV.	Pour aller plus loin.....	28
V.	Critique .....	32
VI.	Conclusion .....	32
VII.	Références.....	33

## I. Introduction

Ce stage avait pour objectif l'analyse de fichiers logs pour essayer de prévoir l'arrivée d'anomalies dans un système. Ce sujet est étudié depuis la fin des années 1990 avec des techniques très basiques étendues en 2005 par des chaînes de Markov [1] ou bien par SVM [2] en 2007 puis très largement en utilisant du machine learning puis du reinforcement learning voir très récemment des modèles de langage avec les modèles GPT [3].

Il existe donc un nombre très important de sources et d'auteurs qui se sont penchés sur le sujet avec des approches très différentes et aux résultats toujours en progression.

Il faut commencer par comprendre ce qu'est un fichier log. Ce type de fichier, au format .log, est généré automatiquement par des logiciels, des systèmes d'exploitation, des applications etc.... Leur sémantique dépend très largement de leur provenance ce qui rend l'analyse très complexe et demande un prétraitement très important, problématique largement prise en compte par tous les auteurs.

### Exemple de log HDFS :

```
081109 203615 148 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_38865049064139660 terminating
081109 203807 222 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_-6952295868487656571 terminating
081109 204005 35 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.73.220:50010 is added to blk_7128370237687728475 size 67108864
081109 204015 308 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_8229193803249955061 terminating
081109 204106 329 INFO dfs.DataNode$PacketResponder: PacketResponder 2 for block blk_-6670958622368987959 terminating
```

Durant ce stage, j'ai décidé de me concentrer sur les logs disponibles librement sur LogHub [4] qui concentre une large variété de logs avec notamment les labels – donnant si le log est une anomalie ou non – présents.

- Apache : HTTP server, est l'un des serveur web le plus populaire,
- BGL : générés par le super ordinateur BlueGene/L en Californie, labelisé,
- HDFS : générés par de très nombreux système physiques, labelisé,
- HPC : générés par le super ordinateur System 20 au Nouveau Mexique,
- Hadoop : utilisé pour faciliter les échanges entre cluster d'ordinateurs, labelisé,
- HealthApp : application mobile pour Android,
- Linux : système d'exploitation, générés par un serveur Linux,
- Mac : système d'exploitation, générés par un Macbook,
- OpenSSH : utilisé pour la connection avec le protocole SSH,
- OpenStack : système d'exploitation cloud pour gérer des datacenters,
- Proxifier : utilisé pour connecter des applications ne supportant pas la communication HTTPS,
- Thunderbird : générés par le super ordinateur Thunderbird au Nouveau Mexique, labelisé,
- ZooKeeper : service de centralisation pour des groupes d'ordinateurs.

En revanche, seuls quatre types de logs sont labelisés, c'est pourquoi seuls ces quatre-là ont été très étudiés dans la littérature.

## II. État de l'art

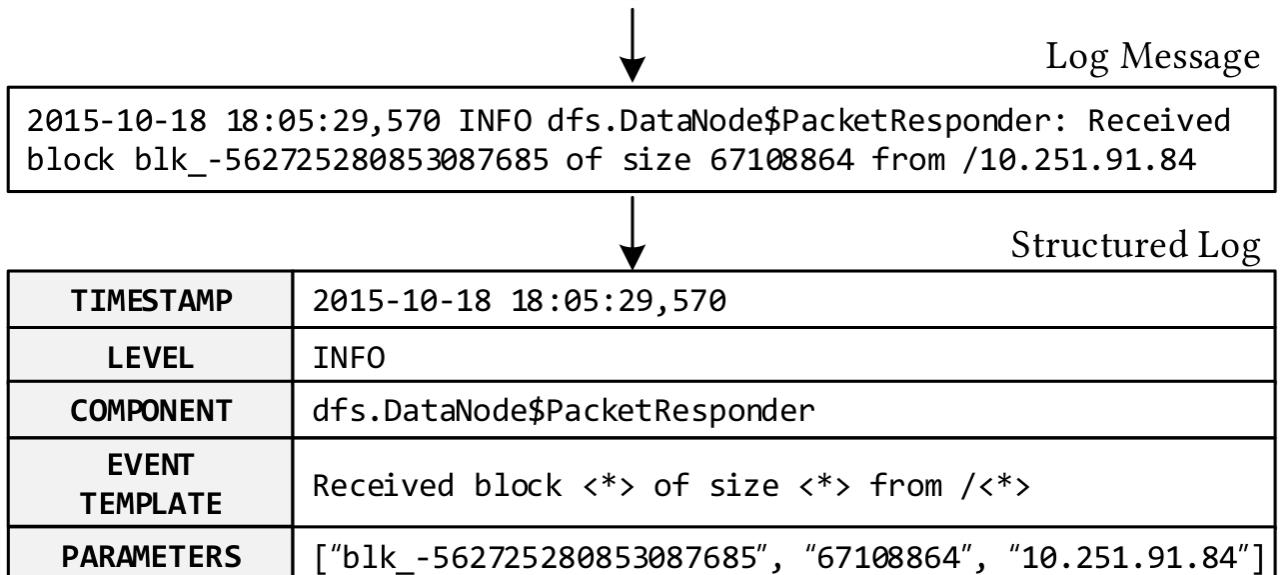
Tout d'abord, il est important de noter qu'il existe trois types de méthodes différentes pour analyser les logs. Comme vu dans l'introduction, certains logs sont labelisés et d'autres non. Ces labels ont été générés à la main par des professionnels et ont nécessité une grande quantité de travail ce qui implique que très peu de logs sont labélisés. Ici, seuls HDFS, Thunderbird, BGL et Hadoop sont labélisés à la différence que BGL et Thunderbird le sont dès leur génération alors que les deux autres nécessitent du travail supplémentaire à la main afin d'en extraire les parties présentant des anomalies.

Ensuite, le fonctionnement de tous les algorithmes repose sur les mêmes principes fondamentaux. Il s'agit de séparer un fichier de logs structurés en deux parties, une qui sert à entraîner le modèle et une qui sert à tester le modèle en vérifiant si la valeur du template présente est la même que celle calculée. La valeur du template correspond aux différents template présents dans le fichier de logs d'origine. Il faut donc une première partie de templatisation : le parsing.

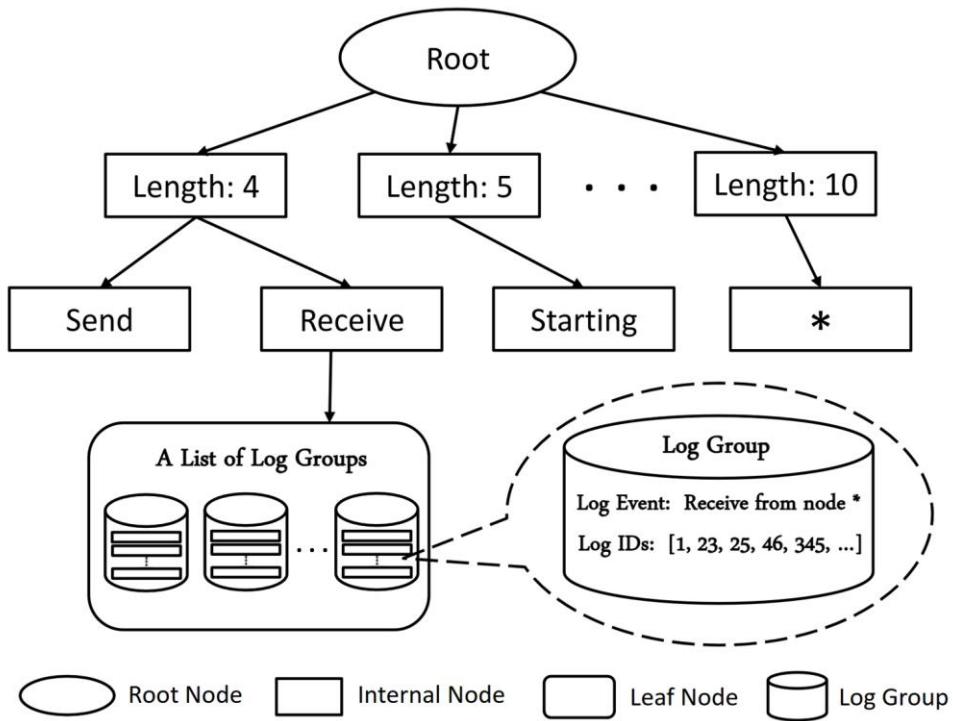
### 1. Templatation (log parsing)

Cette partie, commune à tous les algorithmes, est nécessaire et obligatoire. C'est pourquoi c'est la première à avoir été étudiée. L'objectif est de transformer une séquence de logs et à en extraire un template, une séquence commune à plusieurs lignes et permettant de ne garder que la racine composant le log.

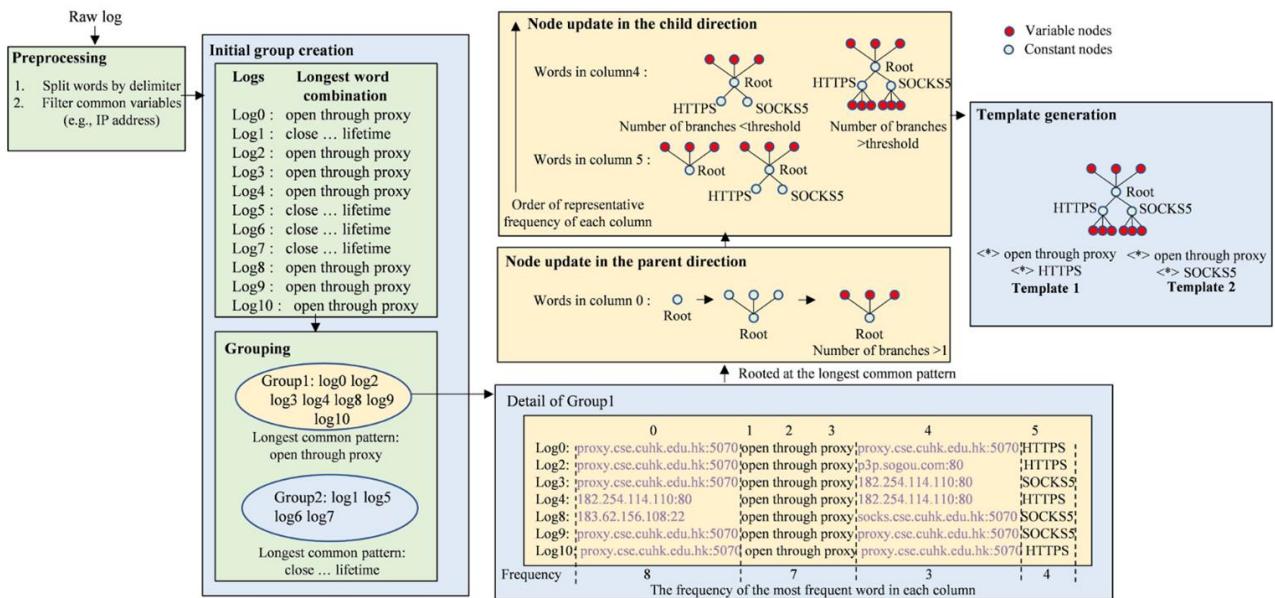
```
/* A logging code snippet extracted from:  
hadoop/hdfs/server/datanode/BlockReceiver.java */  
  
LOG.info("Received block " + block + " of size "  
+ block.getNumBytes() + " from " + inAddr);
```



De nombreux auteurs ont utilisé Drain [6] (2017) pour cette partie préparatoire, basé sur un arbre à profondeur fixe et donnant de très bons résultats.



Plus tard, Brain [7] (2023) tente de faire mieux et réussit grâce à un arbre parallèle bidirectionnel.



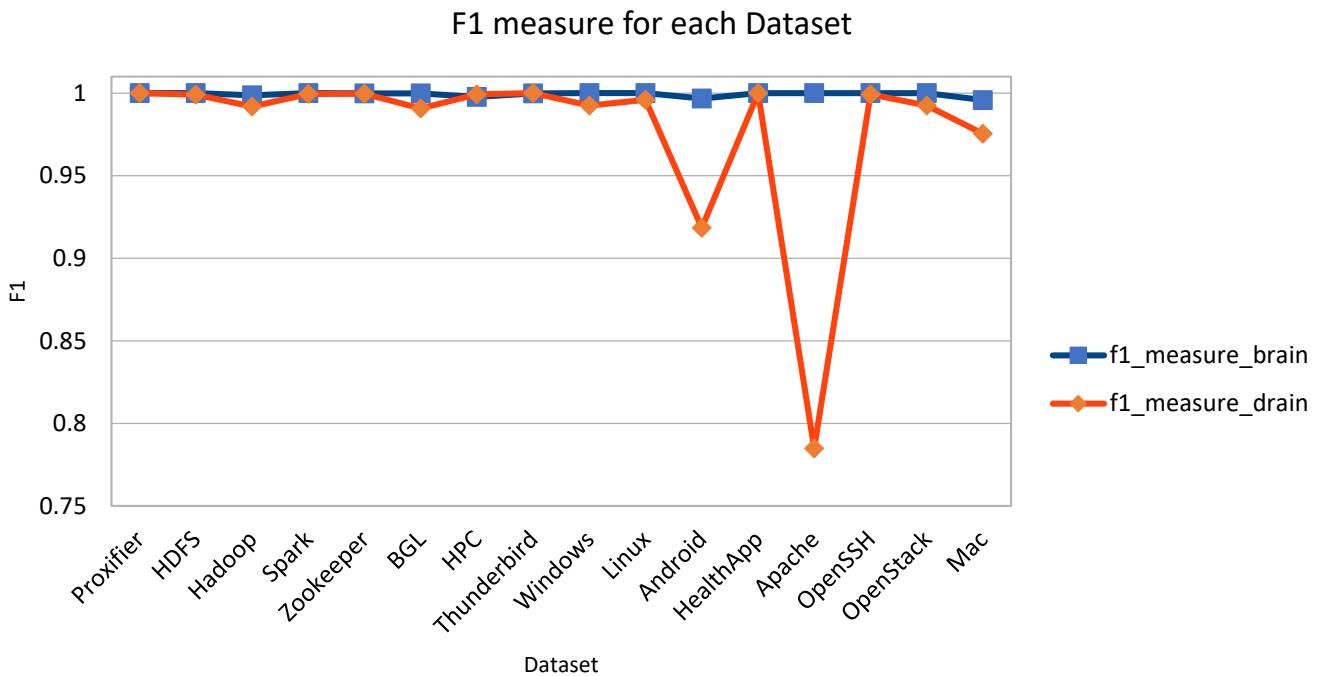
De nombreux autres algorithmes existent pour réaliser cette tâche, il faut donc pouvoir les comparer, c'est ce que LogHub [5] propose.

Dataset	SLCT	AEL	IPLoM	LKE	LFA	LogSig	SHISO	LogCluster	LenMa	LogMine	Spell	Drain	MoLFI	Brain	Best	
HDFS	0.545	0.998	1	1	0.885	0.85	0.998	0.546	0.998	0.851	1	0.998	0.998	0.998	<b>1</b>	Spell
Hadoop	0.423	0.538	0.954	0.67	0.9	0.633	0.867	0.563	0.885	0.87	0.778	0.948	0.957	0.949	<b>0.957</b>	MoLFI
Spark	0.685	0.905	0.92	0.634	0.994	0.544	0.906	0.799	0.884	0.576	0.905	0.92	0.418	0.998	<b>0.998</b>	Brain
Zookeeper	0.726	0.921	0.962	0.438	0.839	0.738	0.66	0.732	0.841	0.688	0.964	0.967	0.839	0.988	<b>0.988</b>	Brain
OpenStack	0.867	0.758	0.871	0.787	0.2	0.2	0.722	0.696	0.743	0.743	0.764	0.733	0.213	1	<b>1</b>	Brain
BGL	0.573	0.758	0.939	0.128	0.854	0.227	0.711	0.835	0.69	0.723	0.787	0.963	0.96	0.986	<b>0.986</b>	Brain
HPC	0.839	0.903	0.824	0.574	0.817	0.354	0.325	0.788	0.83	0.784	0.654	0.887	0.824	0.945	<b>0.945</b>	Brain
Thunderb.	0.882	0.941	0.663	0.813	0.649	0.694	0.576	0.599	0.943	0.919	0.844	0.955	0.646	0.971	<b>0.971</b>	Brain
Windows	0.697	0.69	0.567	0.99	0.588	0.689	0.701	0.713	0.566	0.993	0.989	0.997	0.406	0.997	<b>0.997</b>	Brain
Linux	0.297	0.673	0.672	0.519	0.279	0.169	0.701	0.629	0.701	0.612	0.605	0.69	0.284	0.996	<b>0.996</b>	Brain
Mac	0.558	0.764	0.673	0.369	0.599	0.478	0.595	0.604	0.698	0.872	0.757	0.787	0.636	0.942	<b>0.942</b>	Brain
Android	0.882	0.682	0.712	0.909	0.616	0.548	0.585	0.798	0.88	0.504	0.919	0.911	0.788	0.961	<b>0.961</b>	Brain
HealthApp	0.331	0.568	0.822	0.592	0.549	0.235	0.397	0.531	0.174	0.684	0.639	0.78	0.44	1	<b>1</b>	Brain
Apache	0.731	1	1	1	1	0.582	1	0.709	1	1	1	1	1	1	<b>1</b>	Brain
OpenSSH	0.521	0.538	0.802	0.426	0.501	0.373	0.619	0.426	0.925	0.431	0.554	0.788	0.5	1	<b>1</b>	Brain
Proxifier	0.518	0.518	0.515	0.495	0.026	0.967	0.517	0.951	0.508	0.517	0.527	0.527	0.013	1	<b>1</b>	Brain
Average	0.6297	0.7597	0.806	0.6465	0.6435	0.5176	0.68	0.6824375	0.7666	0.73544	0.7929	0.8657	0.6201	0.9831	N.A.	13/16

Brain est le plus récent et également le meilleur. J'ai donc choisi de l'utiliser pour la suite afin de conserver une base commune à tous les algorithmes que j'ai testé.

Il est important de noter que je n'ai testé sur ma machine que Brain et Drain afin de les comparer directement.

Pour ce faire, j'ai utilisé LogHub et ai fait tourner Brain et Drain pour tous les logs fixés à une longueur de 100.000 lignes dû aux grosses différences de taille entre les fichiers.



Ainsi, j'ai compilé tous les scores F1 pour tous ces types logs sur les deux meilleurs algorithmes afin de les comparer.

Ce graphique montre bien que Brain est bien meilleur que Drain.

Afin de comparer tous les algorithmes de templatisation ainsi que dans la suite de ce rapport, j'utilise le F1-score, calculé comme ceci :

$$F1 = \frac{2 * Precision * Recall}{Precision + recall}$$

Avec,

$$Precision = \frac{TP}{TP+FP} \text{ et } Recall = \frac{TP}{TP+FN}$$

Quid du temps de calcul ?

Test sur HDFS avec 1 million de lignes :

- Drain : 129.0553 secondes
- Brain : 99.0343 secondes

Soit 30 secondes de moins pour un résultat quasiment similaire.

## 2. Détection d'anomalie

Il existe une très grande diversité dans les algorithmes de détections d'anomalies, il faut donc séparer clairement les trois types qui existent :

- Supervisé : utilisation des labels pour l'entraînement et l'application,
- Semi-supervisé : utilisation des labels uniquement pour l'application,
- Non-supervisé : aucune utilisation des labels.

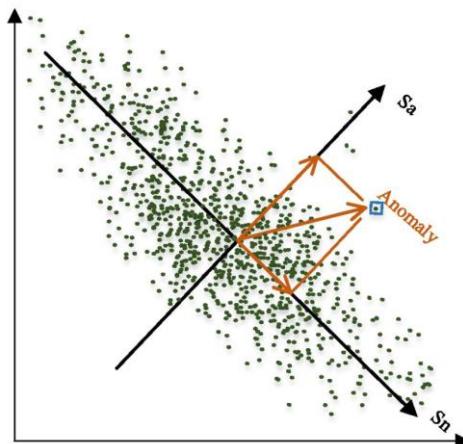
Les algorithmes non-supervisés ont été les premiers expérimentés grâce à leur simplicité mais leurs résultats sont peu fiables et souvent mauvais.

Il faut par ailleurs noter que tous les auteurs se sont principalement intéressés aux logs HDFS de par leur importante présence mais surtout grâce à la labellisation, rendant l'étude et la comparaison plus simple. Les comparer est donc plus simple mais dans les faits, il est très compliqué de les tester sur des types de logs plus réalistes, c'est-à-dire sans label.

### a. Non-supervisé

- PCA – Principal Component Analysis [8]

L'analyse en composante principale est une méthode statistique qui permet de réduire la dimension de données à étudier. L'objectif est de projeter les données dans un nouveau système de coordonnées composé des k composantes principales donnant la meilleure variance (I.E. une droite).



Cette étude datant de 2009, les résultats ne sont pas reproduisibles tels quels. Néanmoins, voici leurs résultats :

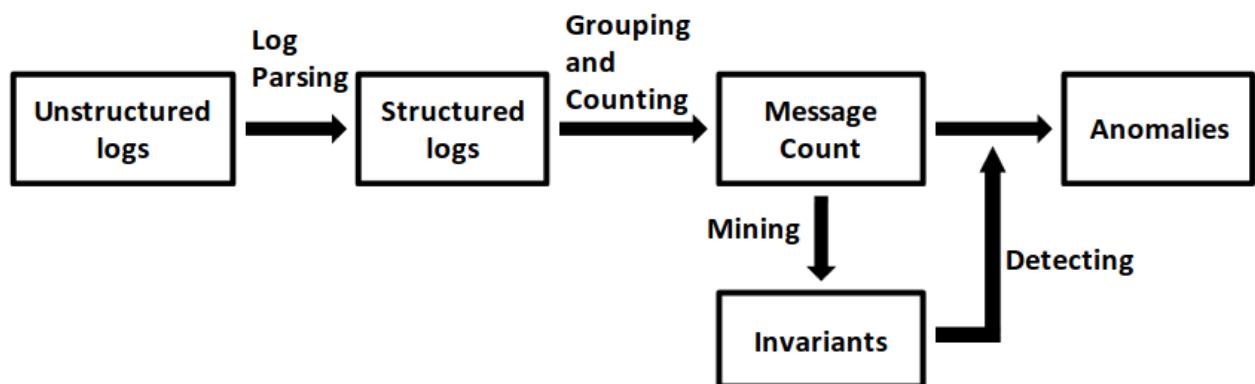
#	Anomaly Description	Actual	Raw	TF-IDF
1	Namenode not updated after deleting block	4297	475	4297
2	Write exception client give up	3225	3225	3225
3	Write failed at beginning	2950	2950	2950
4	Replica immediately deleted	2809	2803	2788
5	Received block that does not belong to any file	1240	20	1228
6	Redundant addStoredBlock	953	33	953
7	Delete a block that no longer exists on data node	724	18	650
8	Empty packet for block	476	476	476
9	Receive block exception	89	89	89
10	Replication Monitor timedout	45	37	45
11	Other anomalies	108	91	107
<b>Total</b>		<b>16916</b>	<b>10217</b>	<b>16808</b>

#	False Positive Description	Raw	TF-IDF
1	Normal background migration	1399	1397
2	Multiple replica (for task / job desc files)	372	349
3	Unknown Reason	26	0
<b>Total</b>		<b>1797</b>	<b>1746</b>

En l'occurrence, testé sur du Hadoop où extraire des composants principaux était plus simple. Dans tous les cas, PCA est utilisé pour extraire les features et donc détecter les anomalies. Ensuite, TF-IDF est utilisé pour alerter un utilisateur d'évidentes anomalies.

### - Invariants Miner [9]

L'objectif de cette méthode est de mettre en avant les similarités entre les logs. Ici, sur HDFS, les « block id » constituent un invariant, toujours présent, dans chaque ligne, quel que soit la taille ou l'origine du log. Par exemple, l'ouverture et la fermeture d'un fichier viendront toujours en paire, l'un à la suite de l'autre. Dès lors, s'il y a un nombre différent d'ouverture et de fermeture, alors il y a une anomalie.



Cette étude datant de 2010, les résultats ne sont pas reproduisibles tels quels.  
Néanmoins, voici leurs résultats :

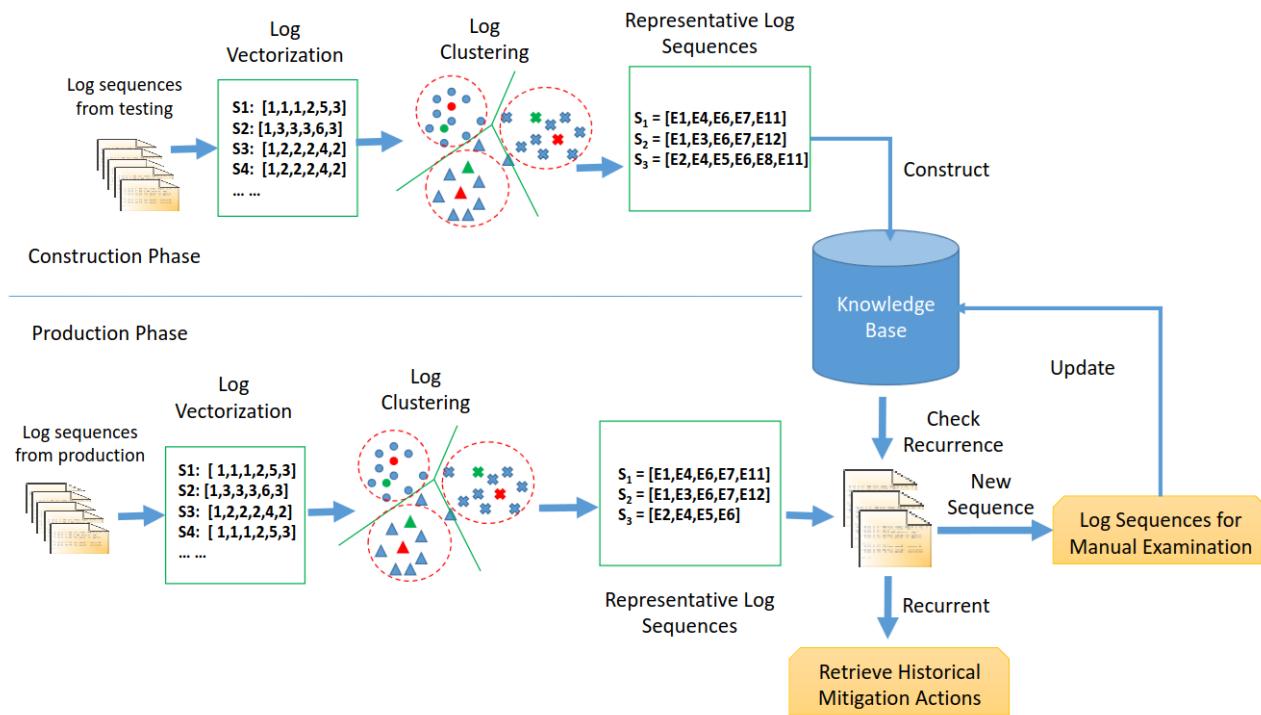
False Positive Description	PCA Method	Our Method
Killed speculative tasks	585	1777
Job cleanup and job setup tasks	323	778
The data block replica of Java execution file	56	0
Unknown Reason	499	0

Là encore, aucun score F1, recall ou bien precision n'est donné. Uniquement une simple comparaison avec la méthode précédente sur la faculté à trouver certaines anomalies, également sur Hadoop.

Anomaly Description	PCA based Method	Our Method
Tasks fail due to heart beat lost.	397	779
A killed task continued to be in RUNNING state in both the JobTracker and that TaskTracker for ever	730	1133
Ask more than one node to replicate the same block to a single node simultaneously	26	26
Write a block already existed	25	25
Task JVM hang	204	204
Swap a JVM, but mark it as unknown.	87	87
Swap a JVM, and delete it immediately	211	211
Try to delete a data block when it is opened by a client	3	6
JVM inconsistent state	73	416
The pollForTaskWithClosed-Job call from a Jobtracker to a task tracker times out when a job completes.	3	3

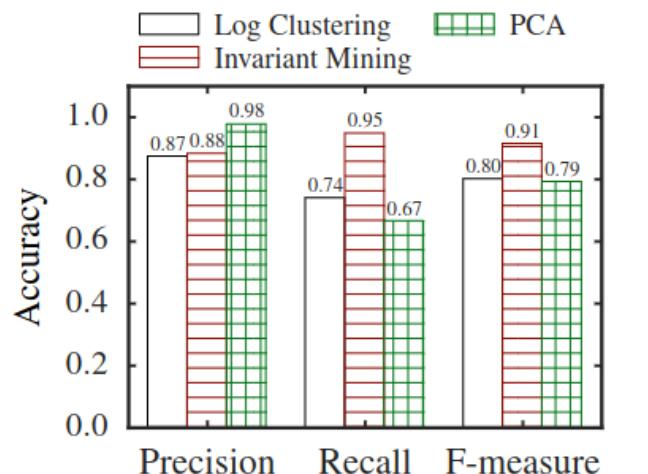
### - LogClustering [10]

Le fonctionnement est basé sur la vectorisation des logs par IDF (Inverse Document Frequency) puis en clusters normaux et anormaux dont le calcul des centroïdes permet de comparer la distance d'une nouvelle séquence de logs à ces centroïdes. Si la distance est supérieure à un seuil, il y a une anomalie.

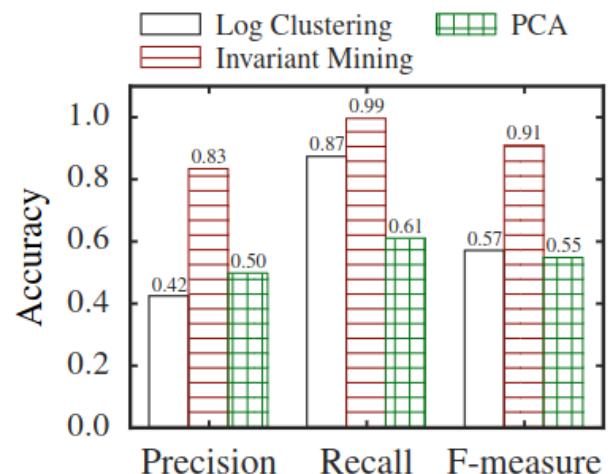


Les deux algorithmes précédents sont en  $O(n)$  ( $n$  le nombres de lignes de logs) mais celui-ci en  $O(n^2)$ .

Loglizer [11] s'est intéressé à comparer ces trois méthodes sur HDFS et BGL.



(a) Accuracy on HDFS data

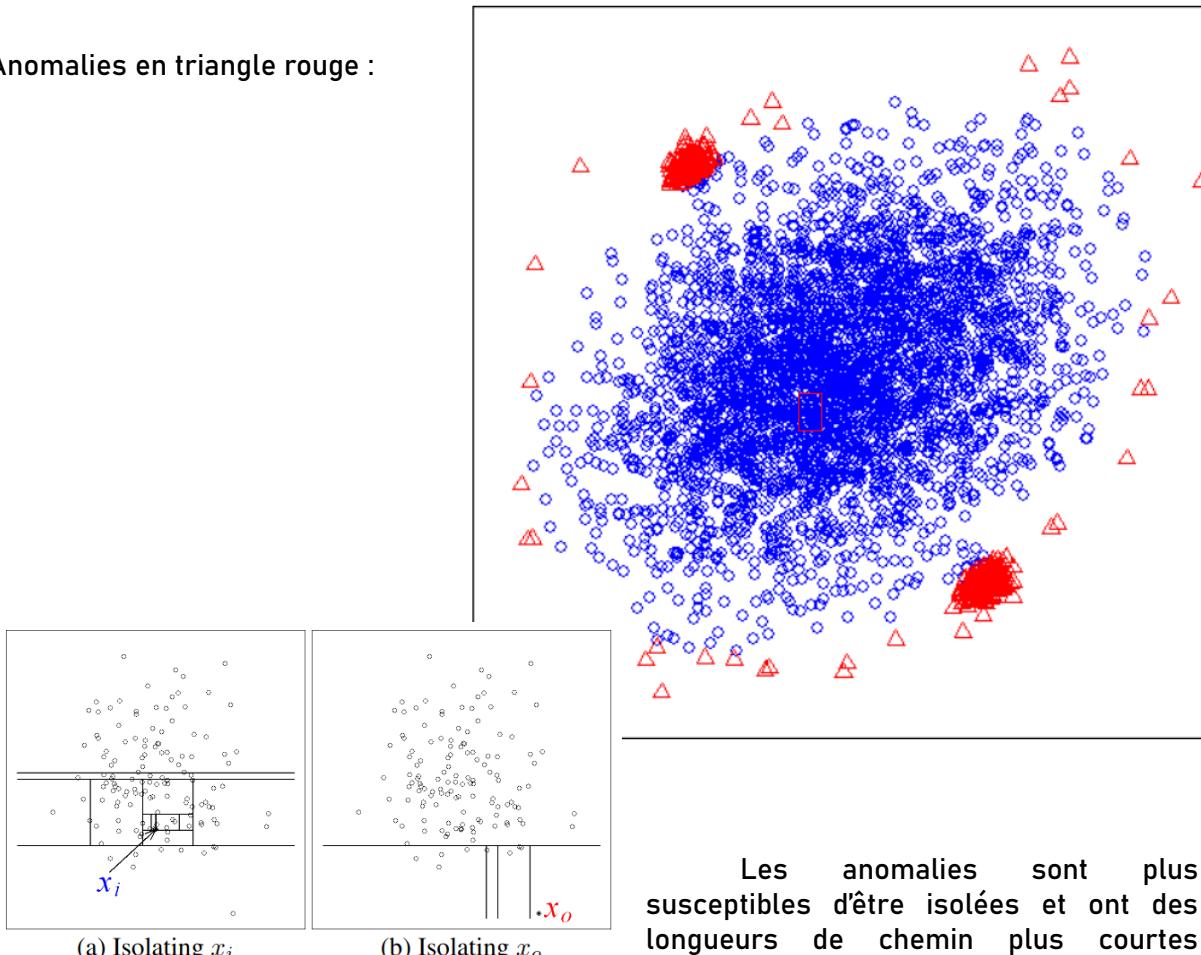


(b) Accuracy on BGL data

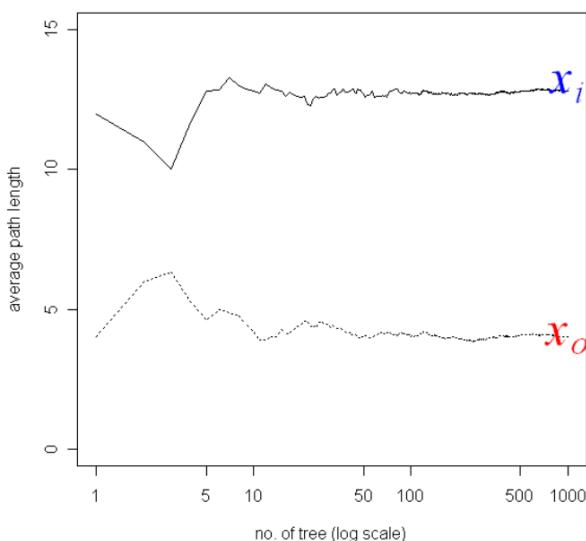
- Isolation Forest [12]

La phase d'entraînement consiste à créer des arbres d'isolement grâce à un set d'entraînement puis de tester la seconde partie des logs à travers des arbres d'isolement pour trouver les anomalies.

Anomalies en triangle rouge :

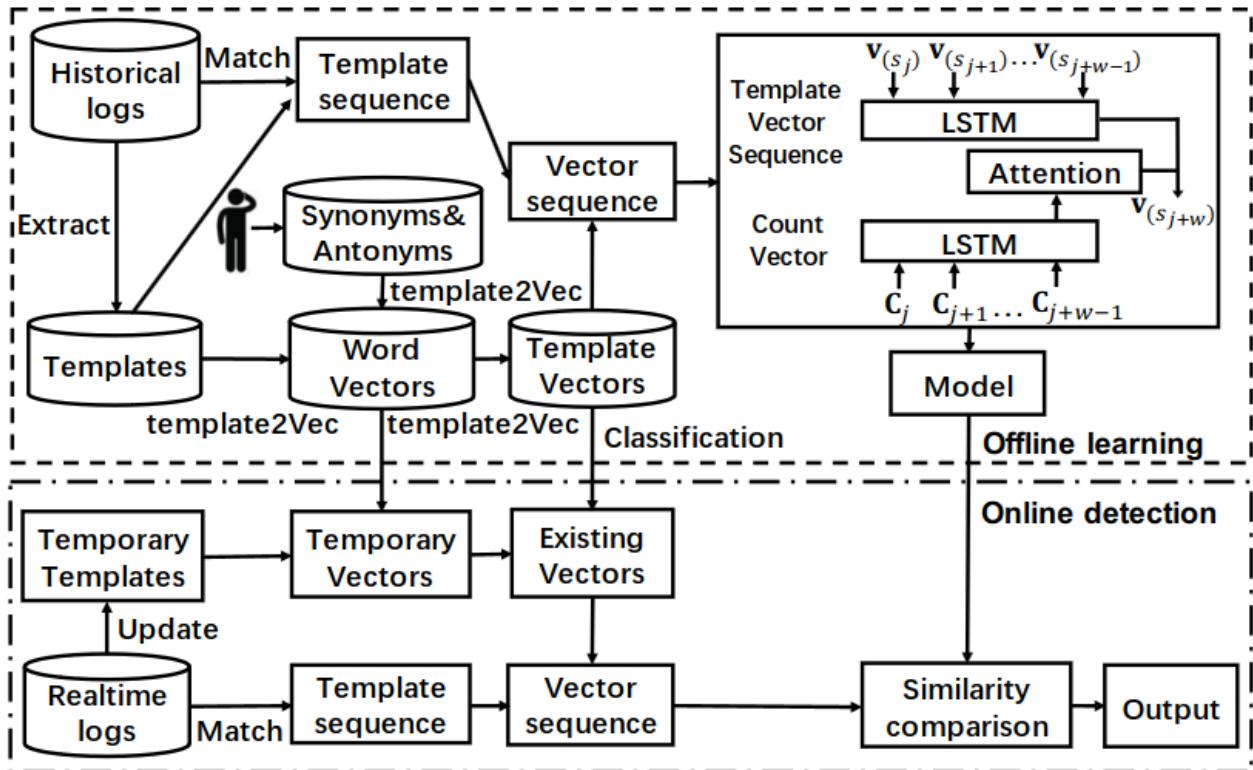


Les anomalies sont plus susceptibles d'être isolées et ont des longueurs de chemin plus courtes (longueur de l'arbre).



Les résultats donnés ne sont malheureusement ni comparables ni reproduisibles.

- LogAnomaly [13]



LogAnomaly utilise un réseau de neurone de type LSTM (Long Short-Term Memory) afin de détecter les anomalies. Pour cela, des templates sont extraits des logs puis sauvegardé durant la phase d'entraînements afin de les comparer durant la phase de test. Les templates sont vectorisés et les features sont extraites et quantifiées grâce au LSTM.

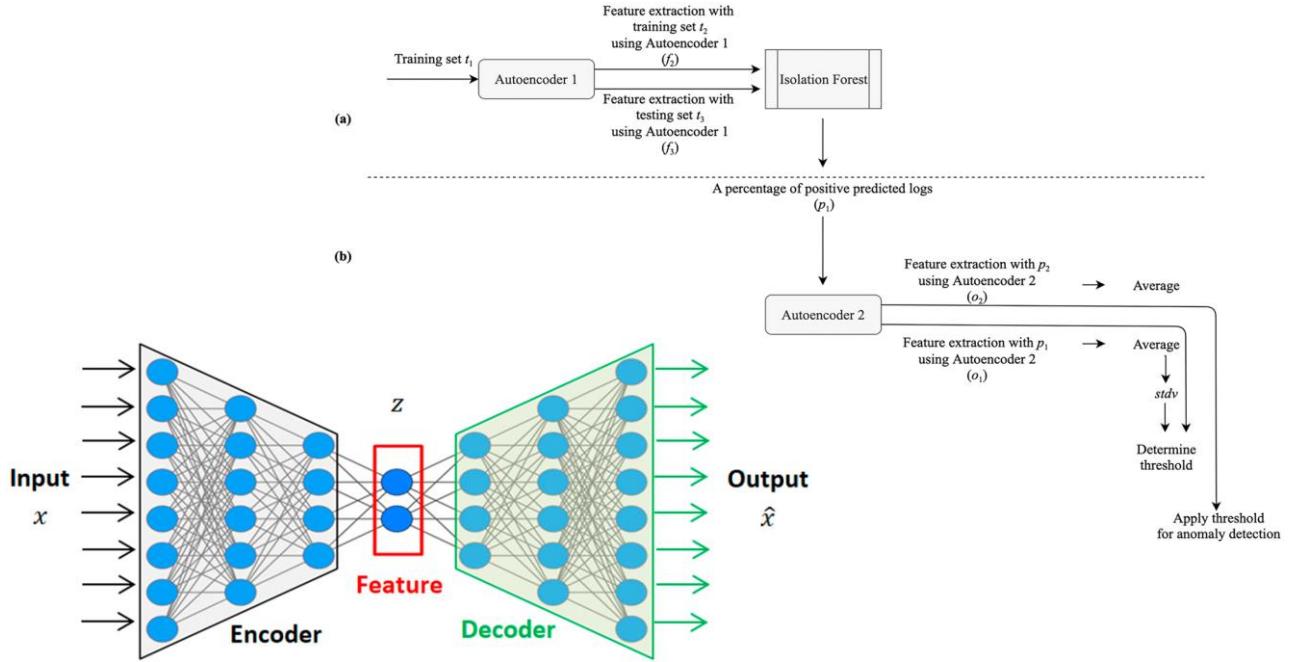
Les auteurs ont testé LogAnomaly sur BGL et sur HDFS :

Score F1 :

- BGL : 96 %
- HDFS : 95 %

- AutoEncoder [14][15]

AutoEncoder est un type de réseau de neurone particulier constitué de deux sous-réseaux : un encodeur et un décodeur, le premier permettant de réduire la dimension de l'ensemble de départ et le suivant en extrait les features importantes afin d'identifier le fonctionnement normal puis les anomalies.



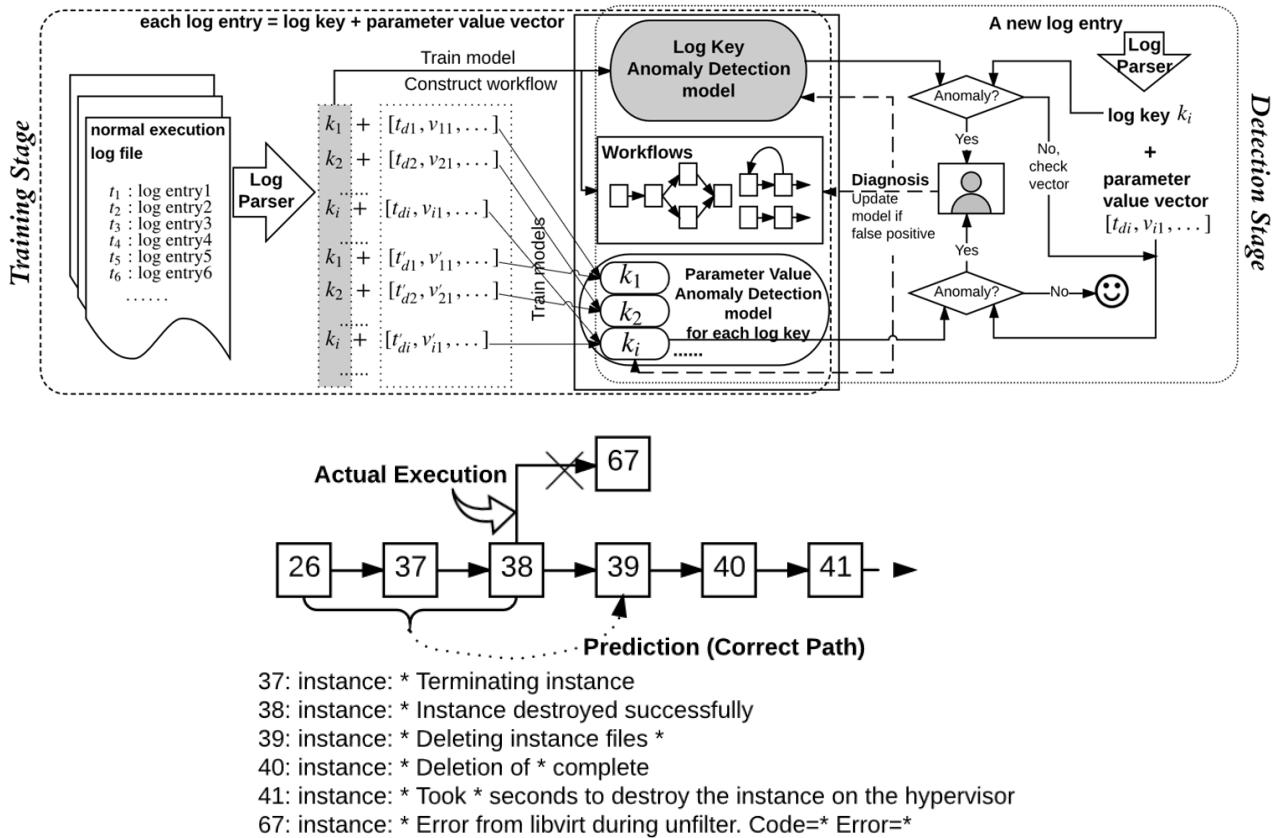
The testing accuracy, precision, recall, F-measure and time for (a) Isolation Forest, (b) Isolation Forest with one Autoencoder and (c) Isolation Forest with two Autoencoders. The minimum, maximum and average (in parenthesis) values for the BGL, Openstack and Thunderbird data sets for 10 runs. Positive labels are denoted by 1 and negative labels by 0.

	Data set	Testing accuracy	Label	Precision	Recall	F-measure	Time (s)
(a)	BGL	88.6%–(88.7%)–88.8%	0	28.1%–(28.9%)–29.7%	35.2%–(36.6%)–38.1%	31.2%–(32.3%)–33.4%	276
			1	94.7%–(94.8%)–95.0%	92.8%–(92.8%)–92.9%	93.7%–(93.8%)–93.9%	
	Openstack	86.9%–(86.9%)–87.0%	0	54.5%–(59.2%)–64.0%	52.5%–(57.6%)–62.8%	53.5%–(58.4%)–63.4%	
(b)	Thunderbird	78.8%–(79.0%)–79.3%	0	29.9%–(30.8%)–31.8%	19.3%–(19.6%)–20.0%	23.5%–(24.0%)–24.6%	158
	BGL	90.6%–(90.8%)–91.0%	0	40.2%–(42.7%)–45.2%	57.1%–(76.5%)–95.9%	47.2%–(54.3%)–61.4%	
			1	96.5%–(98.0%)–99.6%	90.6%–(91.9%)–93.3%	94.9%–(94.9%)–94.9%	
(c)	Openstack	92.9%–(92.9%)–93.0%	0	66.9%–(67.1%)–67.3%	99.4%–(99.5%)–99.7%	80.1%–(80.2%)–80.3%	171
	Thunderbird	81.8%–(82.9%)–84.1%	0	45.6%–(49.9%)–54.2%	41.9%–(49.6%)–57.4%	43.7%–(49.3%)–54.9%	
	BGL	99.4%–(99.6%)–99.9%	0	94.1%–(96.8%)–99.5%	97.8%–(98.7%)–99.6%	96.6%–(98.1%)–99.6%	
	Openstack	98.4%–(99.1%)–99.7%	0	93.5%–(96.1%)–98.7%	95.7%–(97.5%)–99.3%	94.6%–(96.8%)–99.0%	366
	Thunderbird	99.0%–(99.4%)–99.9%	0	94.7%–(97.2%)–99.8%	97.6%–(98.6%)–99.6%	97.2%–(98.4%)–99.7%	
			1	99.6%–(99.7%)–99.9%	98.9%–(99.4%)–99.9%	99.4%–(99.6%)–99.9%	1158

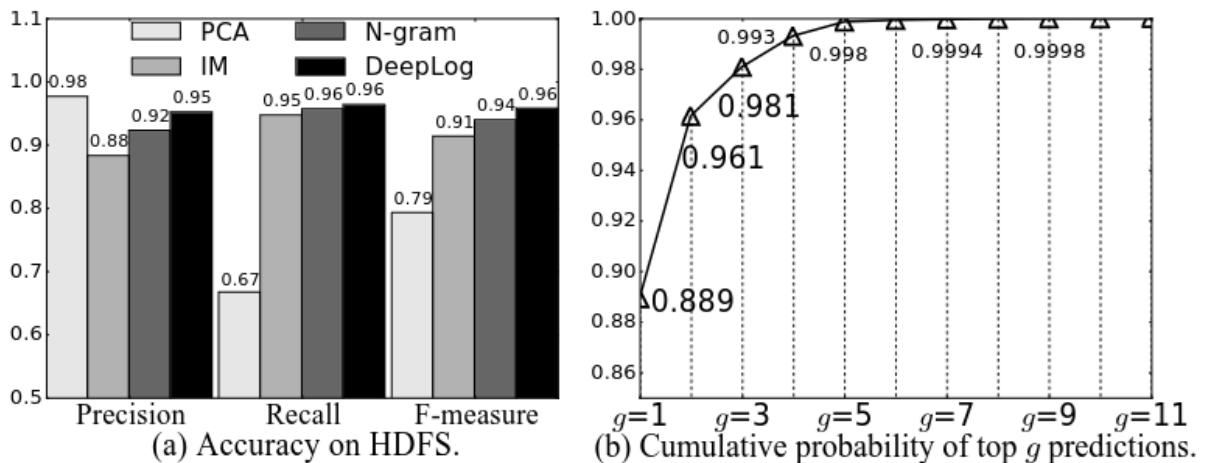
Non-testé sur HDFS mais résultats très bons dès lors que les labels sont présents.

- DeepLog [16]

DeepLog est le premier à utiliser les réseaux de neurones de type LSTM et le Deep-learning pour détecter les anomalies. Le fonctionnement est assez simple, une fenêtre circule sur tous les événements caractérisés par un chiffre et suivant une séquence précise.



Une prediction suit une probabilité et les « top-g » constituent les probabilités les plus importantes.

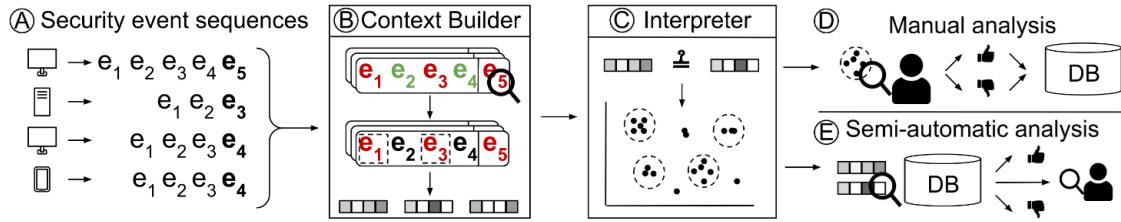


Résultats très bons face aux autres techniques classiques sur du HDFS.

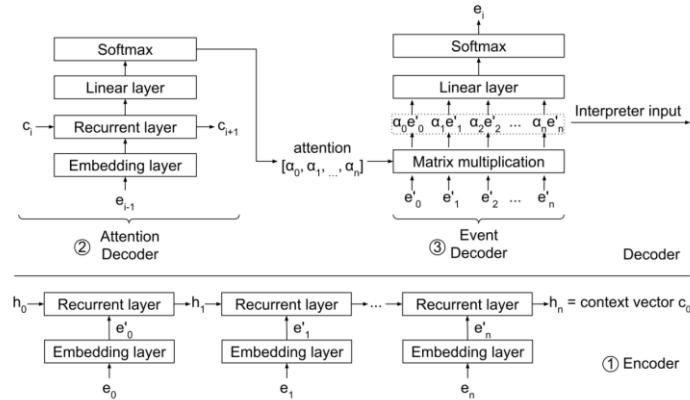
## b. Semi-supervisé

- DeepCase [17]

Basé sur DeepLog, DeepCase va plus loin dans en combinant plusieurs réseaux LSTM mais a pour objectif non seulement de détecter des anomalies, mais surtout de réduire la charge de travail d'un opérateur en lui signalant les problèmes. Les auteurs annoncent une charge de travail réduite de 90 % en n'indiquant que les plus graves problèmes.



Le CONTEXT BUILDER (1) transforme la suite d'évènements en embeddings qui génèrent un vecteur de contexte pris en (2) pour créer un vecteur d'attention puis décrit (3) pour l'interprétation et la détection d'anomalies.

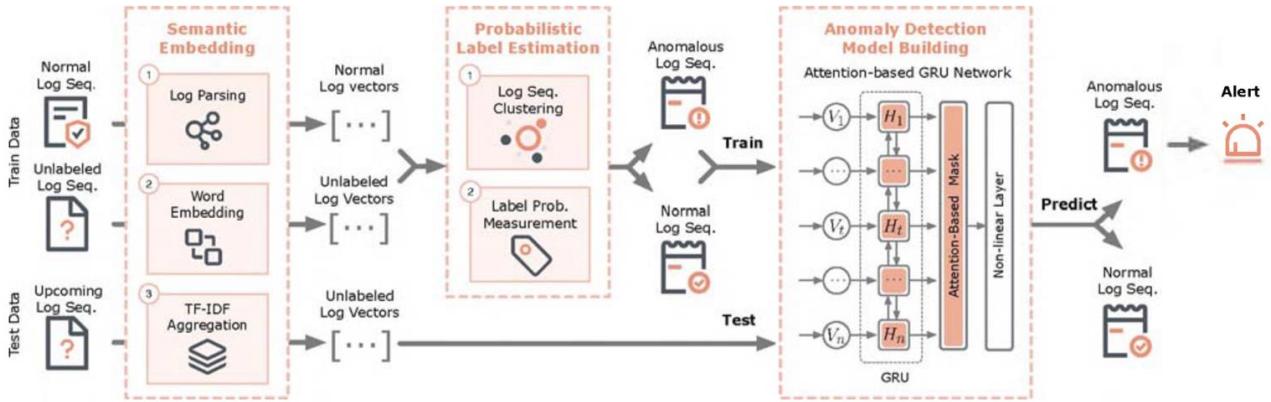


**PREDICTION RESULTS.** SYSTEMS TRAINED ON FIRST 20% OF DATA AND EVALUATED ON REMAINING 80% OF DATA. TIME SHOWS THE AVERAGE AMOUNT OF TIME FOR 1 EPOCH OF TRAINING. BEST PERFORMANCE IS HIGHLIGHTED IN **BOLD**.

	System	Precision	Recall	F1-score	Accuracy	Train time
HDFS	DeepLog	89.71%	89.34%	89.35%	89.34%	<b>1.0 s</b>
	Tiresias	89.70%	87.63%	87.96%	87.63%	15.0 s
	DEEPCASE	<b>90.41%</b>	<b>90.64%</b>	<b>90.40%</b>	<b>90.64%</b>	1.3 s
VMWARE	DeepLog	89.65%	90.40%	89.82%	90.40%	<b>0:06.8 m</b>
	Tiresias	95.50%	96.21%	95.68%	96.21%	4:51.5 m
	DEEPCASE	<b>97.90%</b>	<b>98.06%</b>	<b>97.90%</b>	<b>98.06%</b>	0:13.8 m

## - PLELog [18]

Cette approche semi-supervisée n'utilise qu'une petite partie des labels représentant un fonctionnement normal et labelise la suite des données par des estimations probabilistiques et les estimations réalisées sont utilisées pour détecter les anomalies de la même manière que les techniques supervisées.



La vectorisation est faite grâce à un dictionnaire et utilise l'algorithme FastText [19] puis TF-IDF pour extraire la sémantique des vecteurs.

Un réseau de neurone de type GRU (Unité Récurrente Fermée) permet de faire la détection mais à l'inverse des autres méthodes supervisées binaires (0 – normal, 1 – anomalie), une probabilité est affectée donnant davantage de possibilités et de meilleurs résultats.

Dataset	Method	Precision	Recall	F1-score
HDFS	DeepLog	0.945	0.900	0.922
	LogAnomaly	0.860	0.898	0.878
	LogClustering	<b>1.000</b>	0.836	0.911
	PCA	0.347	0.073	0.121
	PLELog	0.950	<b>0.963</b>	<b>0.957</b>
	LogRobust	0.999	0.995	0.997
BGL	DeepLog	0.138	0.63	0.227
	LogAnomaly	0.179	0.998	0.303
	LogClustering	0.914	0.642	0.754
	PCA	0.448	0.333	0.382
	PLELog	<b>0.965</b>	<b>0.999</b>	<b>0.982</b>
	LogRobust	0.999	0.998	0.999

### c. Supervisé

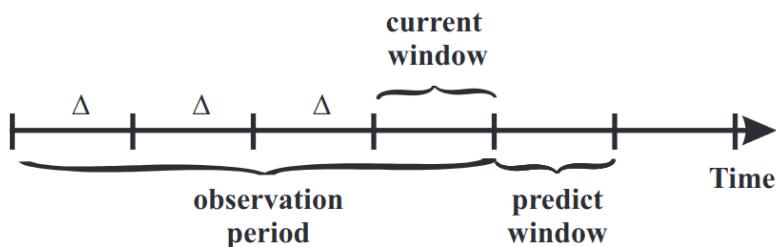
#### - SVM – Support Vector Machine [20]

Dès 2001 [21], les SVM ont été utilisé pour la détection d'anomalies, de manière non-supervisé, sur une seule classe. Par ailleurs, cette méthode est très peu efficace et surtout utilisée pour détecter des chiffres et des lettres dans des images.

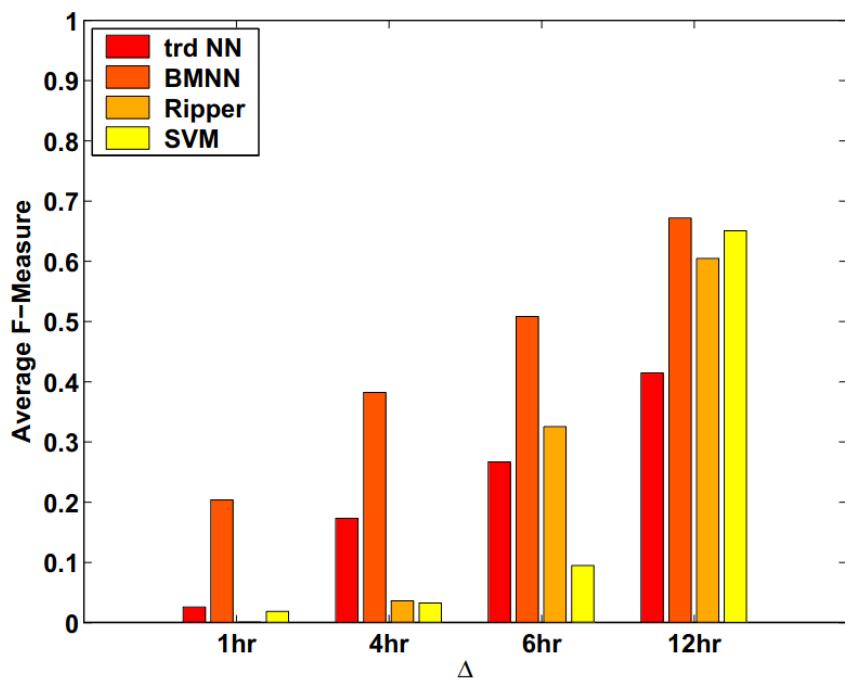
En 2007, un SVM multi-classe est introduit, utilisant les logs labélisés pour détecter des anomalies.

Ici, cinq classes sont utilisées :

- nombre d'évènements présents durant la fenêtre d'étude,
- nombre d'évènements présents depuis le début,
- distribution des évènements sur la fenêtre d'étude,
- intervalle de temps entre deux anomalies,
- fréquence d'apparition d'un évènement durant la fenêtre d'étude.

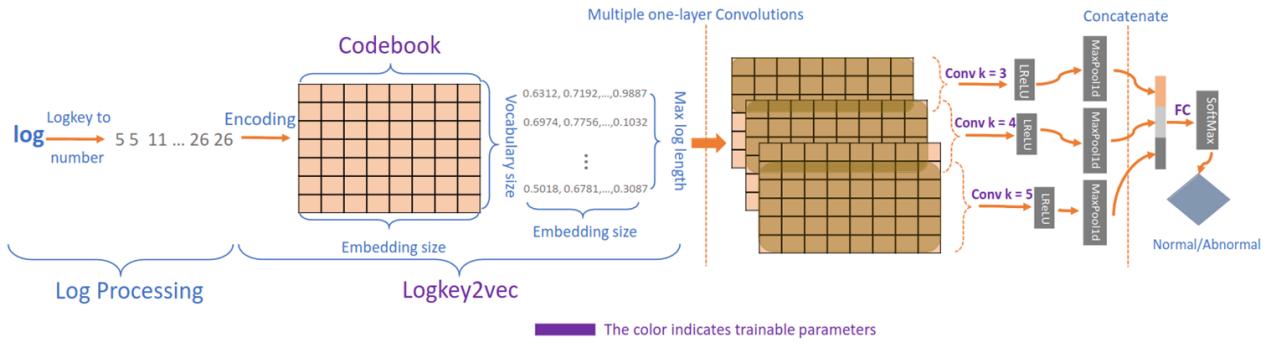


SVM a de bons résultats mais au prix d'un temps de calcul très importants (plusieurs heures). Cette technique a ici été comparé à des techniques de plus proches voisins (Nearest Neighbour - NN).



- CNN – Convolutional Neural Network [22]

Le prétraitement des logs et le même que DeepLog. Les logs structurés sont transformés en sessions par ordre d'exécution avec chaque Blk\_Id (pour HDFS) représentant une session et les numéros de template du log forment une liste dans cette session. Ainsi, la détection d'anomalies est faite sur les sessions correspondantes au Blk\_Id comportant des anomalies connues grâce à un réseau de neurone par convolution.



Afin d'améliorer l'extraction des features, du NLP (Natural Language Processing) est utilisé, ici, Logkey2vec, formant une matrice contenant les embeddings des sessions de logs calculées lors du prétraitement.

Cette technique est également comparée au MLP (MultiLayer Perceptron), un réseau de neurone proche du CNN, et également à un réseau LSTM classique, aux nombres de paramètres plus importants et plus complexes que CNN.

### THE COMPARISON OF DIFFERENT MODELS ON HDFS LOG.

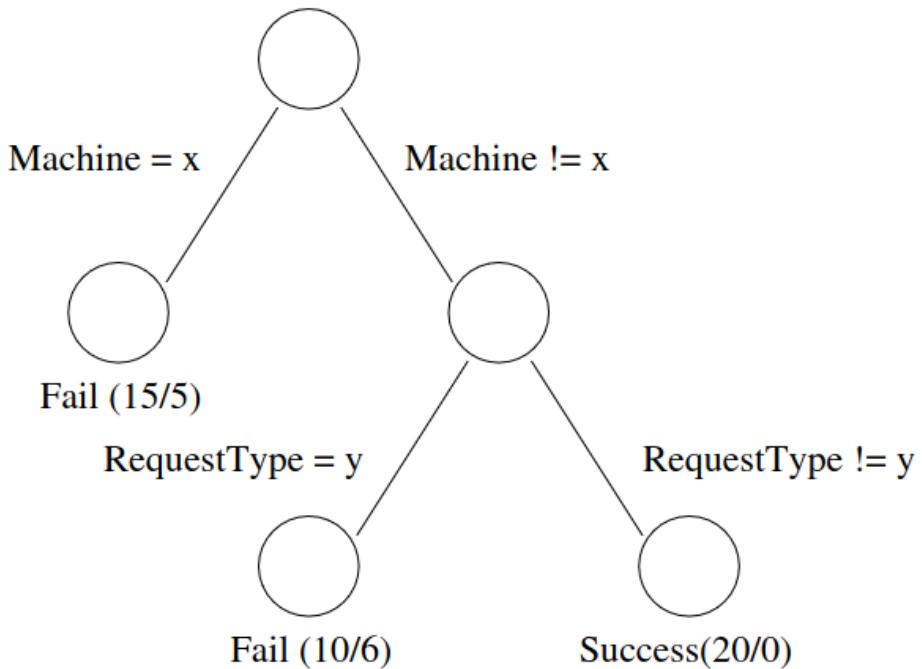
Model	Accuracy (%)	Precision	Recall	F1-measure
CNN	$99.9 \pm 856e-05$	$97.7 \pm 068e-05$	$99.3 \pm 0035$	$98.5 \pm 0014$
MLP	$99.89 \pm 588e-05$	$98.12 \pm 918e-05$	$98.04 \pm 0036$	$98.08 \pm 0018$
LSTM [15]	—	95	95	96

Les résultats sont bons mais ne fonctionnent que sur HDFS.

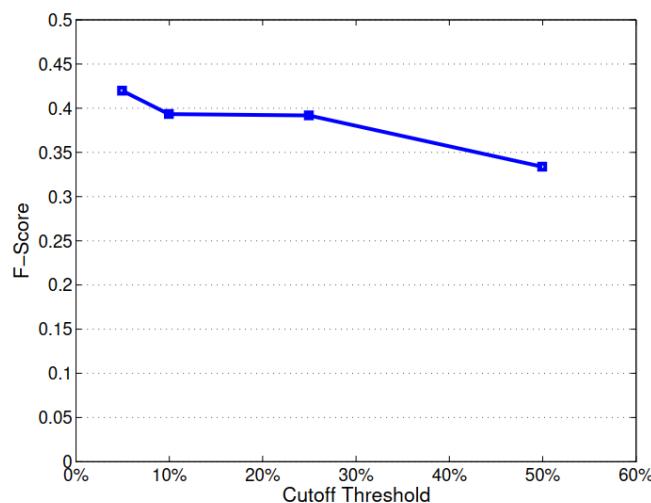
- DecisionTree [23]

Un arbre de décision est construit au fur et à mesure en associant les différents chemins menant à une anomalie grâce aux mêmes sessions que CNN. Il ne s'agit pas d'un réseau de neurone mais vraiment d'une approche stochastique où les anomalies proviennent d'une simple recherche binaire dans un arbre.

Les feuilles qui ne relèvent pas d'anomalies (provenant des Blk\_Id sur HDFS) ne sont pas regardées car ne présentant pas d'information importante. Aussi, les feuilles qui contiennent trop peu d'anomalies (un certain pourcentage seuil, appelé cutoff threshold) sont également ignorées. Enfin, des probabilités sont calculés et des poids associés aux prédictions pour prioriser leur importance.



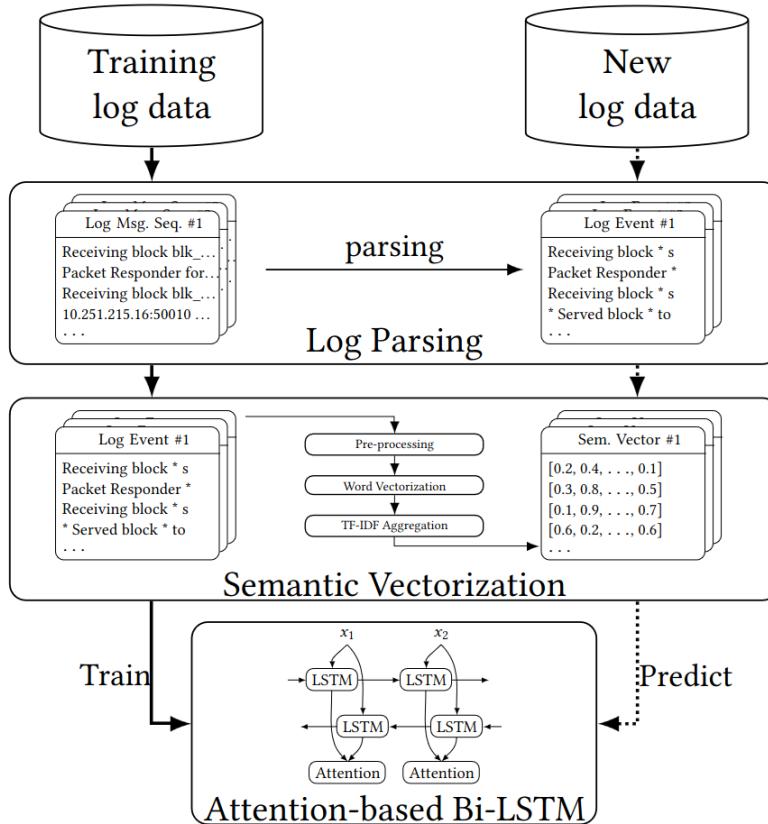
Datant de 2004, les résultats présentés ne sont pas sur HDFS mais un type de log provenant de Ebay, très spécifique.



- LogRobust [24]

Basée sur LogAnomaly et étendant DeepLog, LogRobust utilise un réseau de neurone à LSTM bidirectionnel.

Les logs sont vectorisés par TF-IDF et malgré du bruit (défaux durant la templatisation), la méthode est très robuste et permet de nourrir le réseau basé sur l'attention qui en extrait les features et assigne un degré d'importance à chaque évènement.



FastText [25] est utilisé pour vectoriser les mots et un algorithme TF-IDF (*term frequency-inverse document frequency*) s'ajoute.

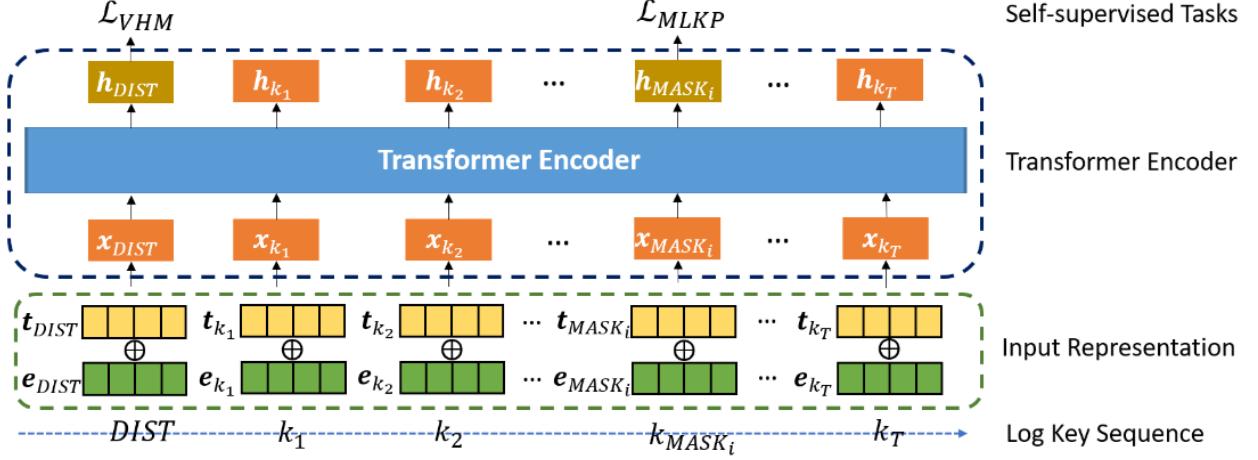
Le LSTM double permet d'annuler complètement les effets secondaires dus à la templatisation et de rendre l'algorithme très robuste.

Method	Precision	Recall	F1-Score
LR	0.99	0.92	0.96
SVM	0.99	0.94	0.96
IM	1.00	0.88	0.94
PCA	0.63	0.96	0.77
<b>LogRobust</b>	0.98	1.00	<b>0.99</b>

Résultats annoncés excellents sur HDFS.

- LogBert [26]

LogBert utilise également la suite d'évènements donnée à la suite de la templatisation. L'entraînement n'est constitué que d'évènements normaux.



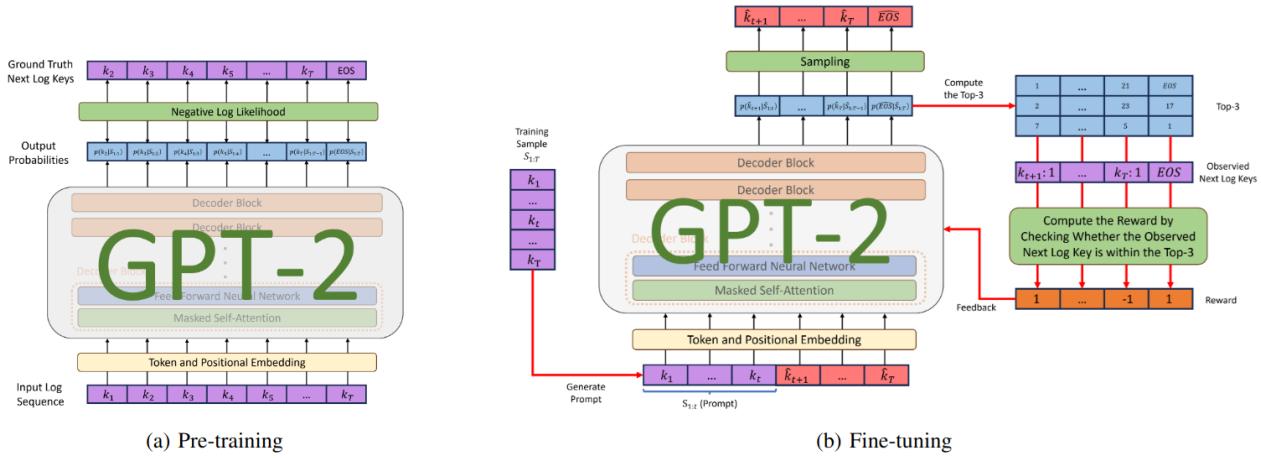
Le Transformer Encoder extrait les features pour que le réseau de neurone apprenne le modèle et détecte les anomalies. Pour ce faire, LogBert utilise un réseau MLKP (Masked Log Key Prediction) et une fonction objectif sert à prédire les évènements futur, masqués durant l'entraînement puis une hypersphère de minimisation pour mieux discriminer les prédictions en minimisant les distances entre la séquence d'entraînement et celle prédite durant la phase de test.

Method	HDFS			BGL			Thunderbird		
	Precision	Recall	F-1 score	Precision	Recall	F-1 score	Precision	Recall	F-1 score
PCA	5.89	100.00	11.12	9.07	98.23	16.61	37.35	100.00	54.39
iForest	53.60	69.41	60.49	99.70	18.11	30.65	34.45	1.68	3.20
OCSVM	2.54	100.00	4.95	1.06	12.24	1.96	18.89	39.11	25.48
LogCluster	99.26	37.08	53.99	95.46	64.01	76.63	98.28	42.78	59.61
DeepLog	88.44	69.49	77.34	89.74	82.78	86.12	87.34	99.61	93.08
LogAnomaly	94.15	40.47	56.19	73.12	76.09	74.08	86.72	99.63	92.73
LogBERT	87.02	78.10	<b>82.32</b>	89.40	92.32	<b>90.83</b>	96.75	96.52	<b>96.64</b>

## - LogGPT [27]

GPT, pour Generative Pre-trained Transformer, est construit sur le même principe que DeepLog est son réseau LSTM en prédisant le prochain évènement. Plus précisément, LogGPT utilise les dernières techniques en LLM (Large Language Model) pour améliorer les performances de la détection.

L'entraînement se base sur le fonctionnement normal pour en extraire et apprendre les sémantiques et le fonctionnement sous-jacent. Une seconde passe vient raffiner le modèle avec du reinforcement learning et les techniques utilisées en LLM, ici GPT-2, créé par OpenAI.



Method	HDFS			BGL			Thunderbird		
	Precision	Recall	F-I score	Precision	Recall	F-I score	Precision	Recall	F-I score
PCA	0.166±0.008	0.059±0.003	0.087±0.002	0.117±0.023	0.035±0.007	0.054±0.010	0.953±0.004	0.980±0.005	0.966±0.003
iForest	0.043±0.010	0.422±0.224	0.078±0.021	0.491±0.364	0.037±0.052	0.063±0.090	0.338±0.128	0.015±0.011	0.028±0.020
OCSVM	0.058±0.012	0.910±0.089	0.108±0.021	0.073±0.003	0.345±0.010	0.121±0.004	0.550±0.004	0.998±0.000	0.709±0.003
LogCluster	<b>0.996±0.003</b>	0.368±0.001	0.538±0.001	<b>0.941±0.015</b>	0.641±0.033	0.762±0.021	<b>0.977±0.005</b>	0.291±0.063	0.445±0.067
DeepLog	0.793±0.092	0.863±0.031	0.824±0.060	0.792±0.048	0.946±0.012	0.861±0.028	0.864±0.005	0.997±0.000	0.926±0.003
LogAnomaly	0.907±0.017	0.369±0.014	0.524±0.017	0.884±0.002	0.850±0.009	0.867±0.003	0.873±0.005	0.996±0.000	0.931±0.003
OC4Seq	0.922±0.059	0.758±0.227	0.808±0.157	0.441±0.045	0.352±0.044	0.391±0.041	0.901±0.046	0.823±0.232	0.845±0.177
LogBERT	0.754±0.142	0.749±0.037	0.745±0.082	0.917±0.006	0.892±0.006	0.905±0.005	0.962±0.019	0.963±0.008	0.963±0.007
CAT	0.102±0.022	0.422±0.082	0.164±0.034	0.177±0.122	0.210±0.184	0.190±0.148	0.751±0.072	0.516±0.124	0.607±0.120
LogGPT	0.884±0.030	<b>0.921±0.066</b>	<b>0.901±0.036</b>	0.940±0.010	<b>0.977±0.018</b>	<b>0.958±0.011</b>	0.973±0.004	<b>1.000±0.000</b>	<b>0.986±0.002</b>

The asterisk indicates that LogGPT significantly outperforms the best baseline at the 0.05 level, according to the paired t-test.

Les résultats de LogGPT sont annoncés comme étant excellents et dépassant toutes les techniques déjà employées.

### III. Résultats

J'ai donc essayé tous ces algorithmes sur ma machine. Il est important de mentionner que les résultats sont très dépendants des composants.

#### 1. Configuration machine

CPU : I7-10700, 8x 2.9 GHz (4.8 GHz)

GPU : Nvidia RTX3060 12Gb

Nvidia Driver : 545.29.06

OS : Linux Ubuntu 22.04 LTS

RAM : 32Gb DDR4

#### 2. Méthode de test

Pour tester ces algorithmes, j'ai utilisé deux dépôts GitHub créés par les auteurs de LogHub (LogPai) qui comparent certains algorithmes, i.e. LogLizer ainsi que DeepLoglizer, ainsi que les GitHub disponibles sur internet pour d'autres. En l'occurrence :

- <https://github.com/logpai/loglizer> pour SVM, DecisionTree, IsolationForest, PCA, InvariantsMining et LogCluster,
- <https://github.com/logpai/deep-loglizer> pour CNN, Autoencoder, LogAnomaly et DeepLog pour HDFS et BGL,
- <https://github.com/Thijsvanede/DeepLog> pour DeepLog,
- <https://github.com/Thijsvanede/DeepCASE> pour DeepCase,
- <https://github.com/HelenGuohx/logbert> pour LogBert,
- <https://github.com/nokia/LogGPT> pour LogGPT,
- <https://github.com/LeonYang95/PLELog> pour PLELog,
- <https://github.com/LogIntelligence/LogADEmpirical> pour LogRobust.

Je me suis d'abord intéressé à HDFS, utilisé par toutes les techniques mentionnées car il comporte les labels permettant de détecter les anomalies bien plus précisément.

J'ai commencé mes tests sur DeepLog mais les données en entrée sont assez sombres, rien n'est expliqué sur quoi lui donner en entrée. En effet, la version de Van Ede ne prend pas en entrée le log structuré sortant de la templatisation mais autre chose. En cherchant un peu, j'ai déterminé qu'il faut donner en entrée un fichier csv contenant trois colonnes : timestamp, machine et event. J'ai donc dû écrire une fonction python qui prend en entrée n'importe quel fichier csv généré suite à la templatisation et qui crée le fichier csv compatible avec DeepLog. De cette façon, j'ai obtenu des résultats sur tous les types de logs mentionnés en première partie.

Ensuite, j'ai testé la même chose sur DeepCase, capable de prendre le même type de fichier mais n'ai pas abouti, DeepCase étant plus complexe.

J'ai ensuite testé tous les autres de la façon dont ils étaient décrits sur leur GitHub respectifs.

Sur HDFS, tout se passe bien et j'obtiens les résultats présentés dans le tableau en fin de partie.

BGL a également été beaucoup étudié mais beaucoup moins que HDFS. J'ai donc testé BGL sur les algorithmes qui le propose nativement, également avec Thunderbird, ces types de logs proposant une sémantique très proche, ne nécessitant pas de traitement supplémentaire.

Par contre, pour tous les autres, j'ai dû adapter ou recoder les algorithmes permettant de charger ce type de logs et de faire les recherches d'anomalies.

Voici donc les scores F1 cumulés dans un tableau pour les algorithmes mentionnés plus haut et présentés dans la partie précédente ainsi que pour les types de logs présentés en première partie. Il est important de noter que ce tableau n'est pas complet :

Supervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (16M)
LogRobust	96.00%	//	90.00%	//	//	//	//	//	//	//	//	97.01%	
CNN	95.59%	//	90.00%	//	//	//	//	//	//	//	//	96.23%	
LogBert	77.81%	//	XX	//	//	//	//	//	//	//	//	89.47%	
LogGPT	99.33%	//		//	//	//	//	//	//	//	//	99.13%	98.99%
SVM	91.08%	//	88.00%	//	//	//	//	//	//	//	//	64.10%	
DecisionTree	99.53%	//	86.96%	//	//	//	//	//	//	//	//	16.34%	82.40%
Semi-supervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (10M)
DeepCase	89.34%		43.35%									XX	
PLELog	97.35%												88.91%
Unsupervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (10M)
PCA	41.92%	100.00%*	91.49%	100.00%*		100.00%*	100.00%*	100.00%*	100.00%*	100.00%*	100.00%*	40.20%	
DeepLog	84.71%	88.56%	42.04%	81.34%	12.86%	21.74%	14.49%	62.45%	73.35%	99.06%	64.05%	86.31%	
LogAnomaly	31.50%											XX	
AutoEncoder	86.42%											77.15%	
LogCluster	87.49%	100.00%*	100.00%*	100.00%*		100.00%*	100.00%*			100.00%*		89.80%	XX
Iforest	38.14%	100.00%*	100.00%*	100.00%*		100.00%*	100.00%*			100.00%*		32.50%	XX
InvariantsMiner	14.06%	100.00%*	100.00%*	100.00%*		XX	100.00%*			XX		41.12%	

\* : overfitting, // : non labélisés, XX : impossible, >>> : trop long.

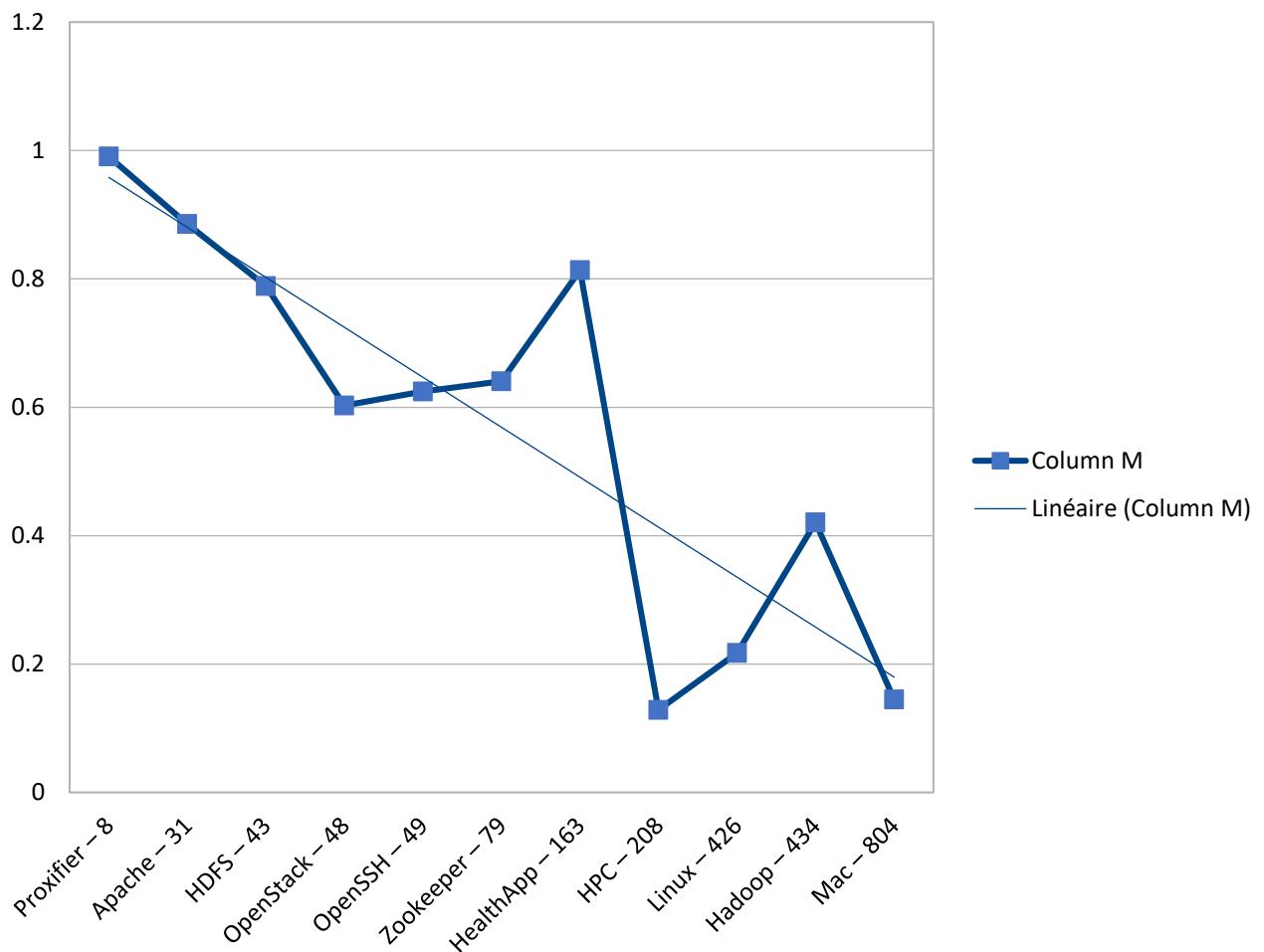
Et les temps de calcul :

Supervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (16M)
LogRobust	63:35min	//	1sec	//	//	//	//	//	//	//	//	77sec/epoch	
CNN	39:35min	//	1sec	//	//	//	//	//	//	//	//	31sec/epoch	
LogBert	73:57+9.52min	//	XX	//	//	//	//	//	//	//	//	15:00+2:17min	
LogGPT	5:20min	//		//	//	//	//	//	//	//	//	18:52min	20:03min
SVM	6:30min	//		//	//	//	//	//	//	//	//	50sec	
DecisionTree	6:20min	//		//	//	//	//	//	//	//	//	57sec	4:12min
Semi-supervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (10M)
DeepCase	1:50min/epoch											XX	
PLELog	17:24min												5:45min
Unsupervised													
Nom	HDFS	Apache	Hadoop	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Zookeeper	BGL	Thunderbird (10M)
PCA	6:40min	0.6sec	0.1sec	0.1sec	.5sec	.3sec	3.8sec	0.27sec	0.27sec	41sec			
DeepLog	36:12min	1sec/epoch	5.5sec/epoch	5.9sec/epoch	39sec/epoch	2sec/epoch	4.3sec/epoch	16sec/epoch	10sec	.5sec/epoch	4sec/epoch	78sec/epoch	XX
LogAnomaly	6:30+8:15min											77sec/epoch	
AutoEncoder	13:12min												
LogCluster	6:38min	14sec		7sec		14sec	3:37min			8sec		8:10min	XX
Iforest	6:25min	29sec		42sec		32sec	11:05min			14sec		14:37min	XX
InvariantsMiner	10:15min	6:13min			47:50min	>>>	29:08min			>>>		97:40min	

A noter que de nombreuses valeurs manquent pour InvariantsMiner car les temps de calcul ont parfois excédé les 8h, soit la fin de journée.

Toutes les valeurs à 100 % dans les non-supervisées ne sont pas correctes, en effet, il s'agirait d'overfitting, c'est-à-dire que le modèle calque trop parfaitement sur les données au lieu d'extraire ses caractéristiques. Par ailleurs, regarder les temps de calcul devient plus pertinent. En effet, on remarque facilement que InvariantsMiner est très long pour des résultats relativement faibles tandis que LogCluster semble donner des résultats intéressants malgré un temps de calcul très variable.

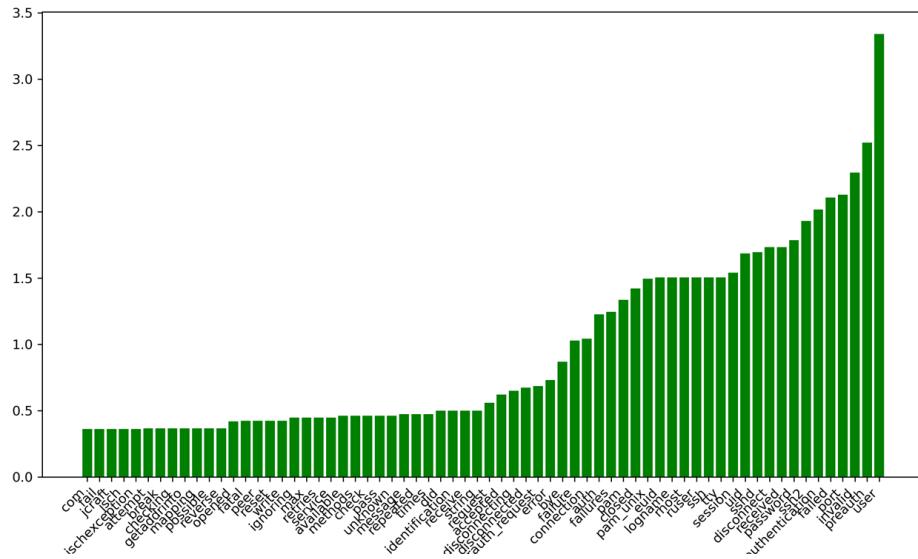
Enfin, je trouve intéressant de s'arrêter un peu sur DeepLog dont le fonctionnement est le plus simple et les résultats globalement corrects, avec des score F1 plutôt bons et des temps de calculs contenus. En revanche, j'ai noté une grande différence dans le score si le nombre de templates présents dans le log est plus importants :



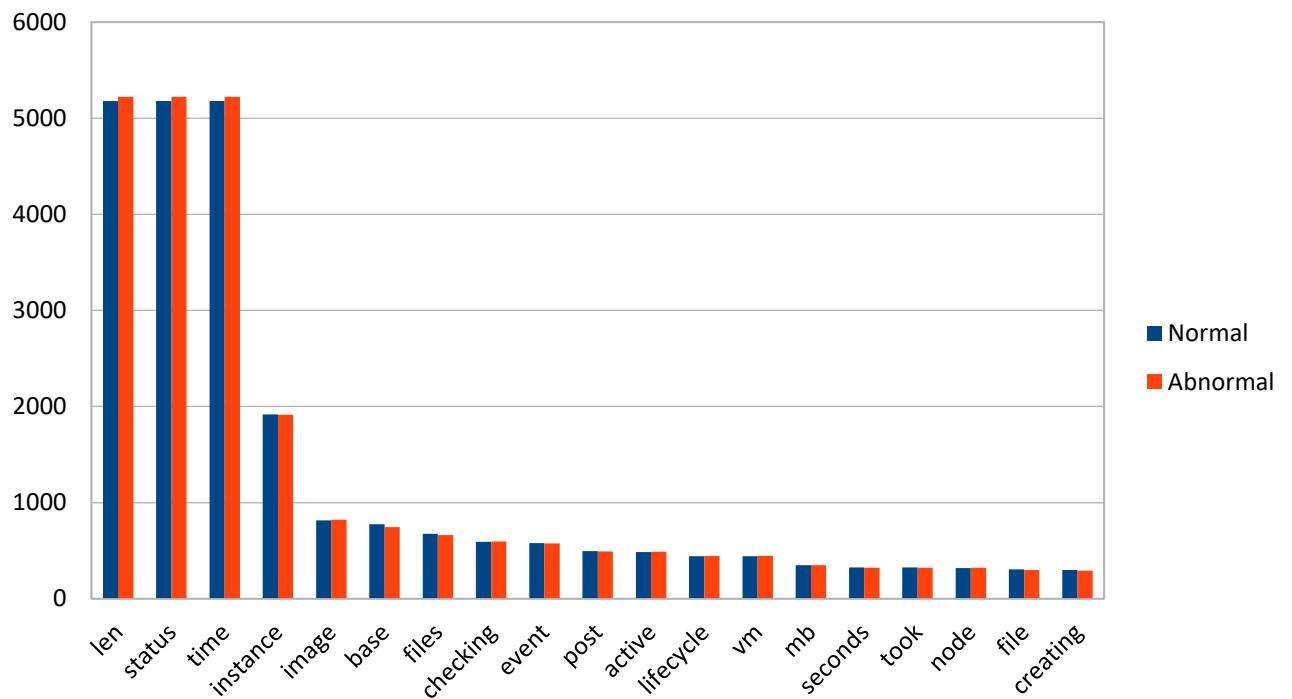
Néanmoins, par manque de temps, je n'ai pu remplir que partiellement le tableau. Je remarque tout de même que malgré la technique utilisée, des méthodes supervisées fonctionnent toujours mieux et tous les algorithmes ont des complexités de l'ordre de  $O(n^2)$  pour les meilleurs, ce qui les rend très lent sur des logs plus longs. Aussi, toutes les méthodes non-supervisées ont été codées en single-thread et sont donc plus lentes.

## IV. Pour aller plus loin

Par curiosité, j'ai voulu regarder ce que l'algorithme de TF-IDF pouvait donner et on voit bien des différences entre certaines termes, dans leur fréquence d'apparition. Il serait intéressant de voir les différences entre un fonctionnement avec et sans anomalies.



LogHub permet de télécharger les logs sur OpenStack, séparés entre un fonctionnement normal et abnormal. Dès lors, en appliquant l'algorithme TF-IDF, on voit apparaître quelques différences :



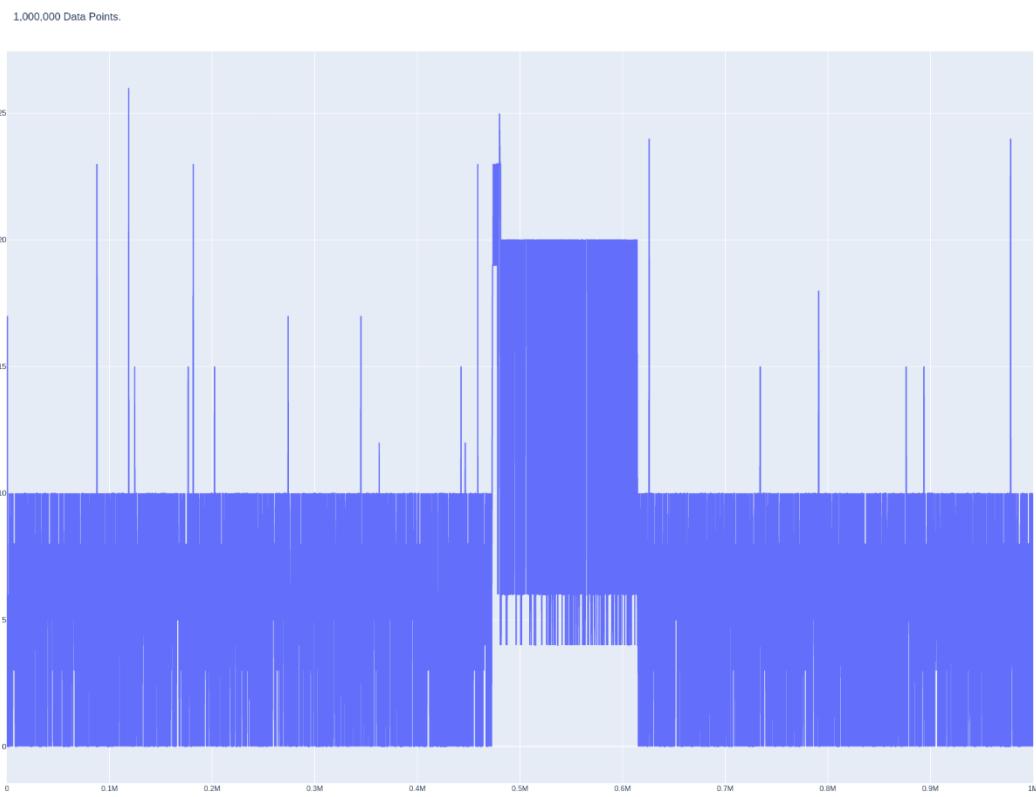
On observe que la fréquence d'apparition diffère légèrement mais ce graphique ne montre pas l'ordre des termes, présentant bien plus de différences :

Abnormal		Normal	
len	5221.45082711466	len	5180.32806117141
status	5221.45082711466	status	5180.32806117141
time	5221.45082711466	time	5180.32806117141
instance	1915.18233309954	instance	1916.4008792693
image	822.119522806238	image	816.354772697177
base	745.095849512028	base	774.552772282506
files	661.171073442882	files	674.074350821192
checking	596.574690007025	checking	591.817693069377
event	577.101792670112	event	577.384762577218
post	492.639682671804	post	494.315093057851
active	487.338695922132	active	486.808708664264
lifecycle	443.896248502211	lifecycle	443.532020541344
vm	443.896248502211	vm	443.532020541344
mb	349.188525306514	mb	349.187525149247
seconds	323.714193211536	seconds	324.542804438873
took	323.714193211536	took	324.542804438873
node	320.909163684282	node	318.471969159705
creating	300.486614384465	file	306.64064189303
local	293.005656763456	creating	300.415935503466
nodes	293.005656763456	local	290.729656576085
sharing	293.005656763456	nodes	290.729656576085
storage	293.005656763456	sharing	290.729656576085
use	293.005656763456	storage	290.729656576085
file	279.474375461522	use	290.729656576085
vcpu	266.379232059169	vcpu	266.381371654076
total	262.066860103132	total	261.554560784337
successfully	235.833112593281	successfully	232.110532752399
limit	231.678672313511	limit	231.686679364856
resumed	225.181287800038	resumed	225.178076101848

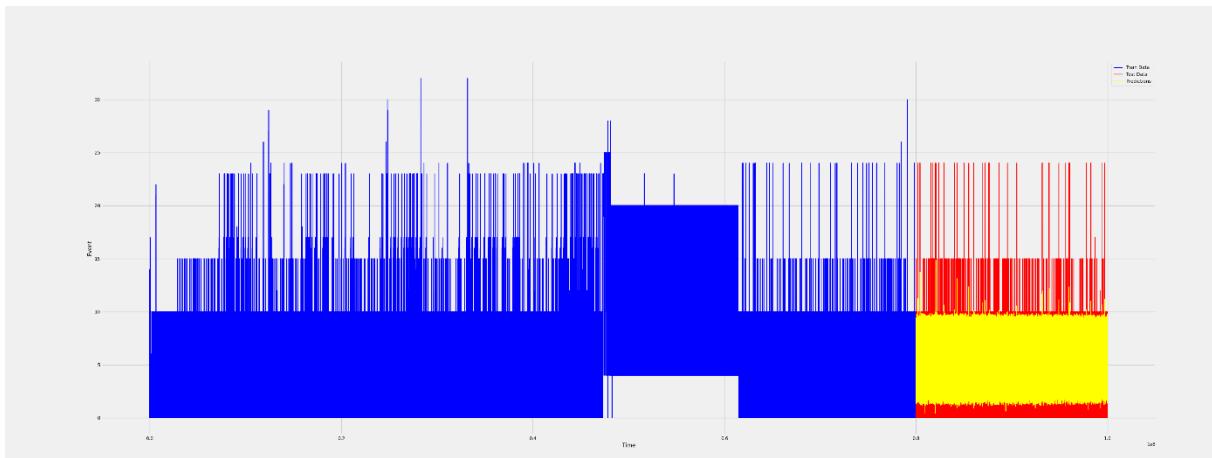
Sur les termes dont la fréquence est la plus faible, il y a encore plus de différences, ce qui est normal. Même sur un autre jeu de données normales, les résultats seront différents.

Ainsi, peut-être que cette feature pourrait être utilisée, ainsi que l'ordre, pour un classifieur tel que SVM avec plusieurs classes pour identifier les anomalies. Le modèle devrait être entraîné sur davantage de données dont le fonctionnement normal ou anormal est connu.

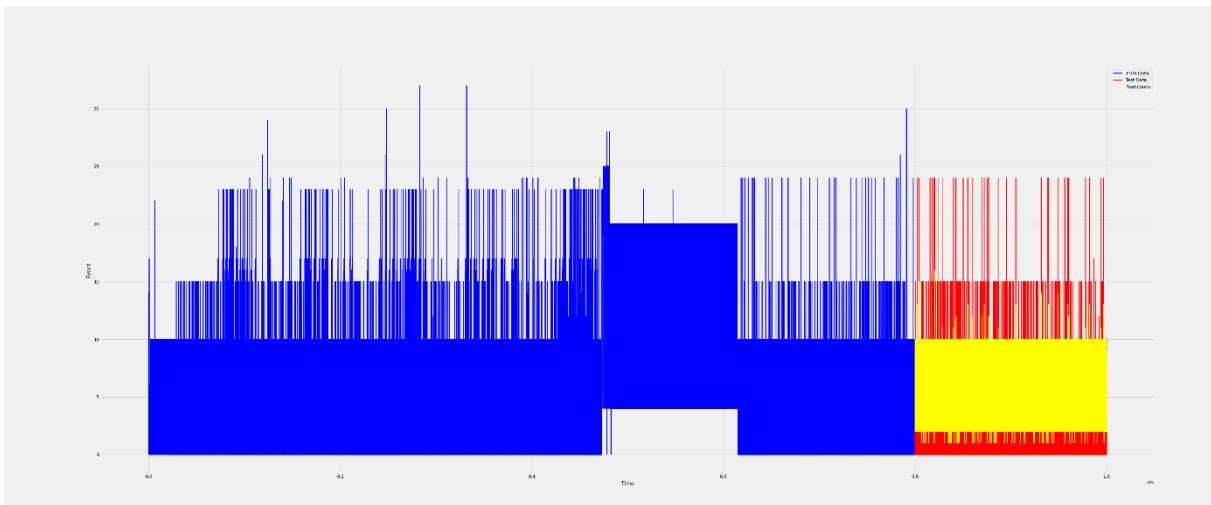
Aussi, j'ai essayé de regarder une série temporelle générée avec le même algorithme que pour créer le fichier csv compatible avec DeepLog donc avec le temps en abscise et l'EventId en ordonnée.



En appliquant un modèle à LSTM produit par la bibliothèque Keras de Python, on peut l'entraîner sur 80 % des données et tester sur les 20 % restants.



Les résultats sont par ailleurs des flottants.



Ainsi, on remarque que les évènements les plus présents sont bien prédis.

Event	EventTemplate	Occurrence
E0	('Receiving', 'block', '<*>', 'src:', '<*>', 'dest:', '<*>')	196869
E10	('BLOCK**', 'NameSystem.addStoredBlock:', 'blockMap', 'updated:', '<*>', 'is', 'added', 'to', '<*>', 'size', '<*>')	194461
E5	('PacketResponder', '<*>', 'for', 'block', '<*>', '<*>')	194257
E8	('Received', 'block', '<*>', 'of', 'size', '<*>', 'from', '<*>')	194245

Le 0 n'est par ailleurs jamais atteint ni tous les évènements dont la valeur est plus importante mais le nombre d'occurrence le plus faible.

Pour analyser le résultat, on peut regarder l'erreur quadratique moyenne, ce qui donne, après avoir trouvé les meilleurs paramètres pour le réseau, une valeur de 6.48, ce qui est loin d'être excellent.

Une cause de cette erreur élevée peut provenir des très gros pics, très présents mais l'information est contenue majoritairement dans les évènements de valeurs plus faible, comme vu précédemment avec les occurrences. Dès lors, les changements très importants sont difficiles à prédire, ce type de modèle fonctionne bien mieux lorsque les changements sont moins rapides et qu'il y a visuellement une périodicité dans les données. Celle-ci n'est très clairement pas présente dans les logs au niveau macroscopique mais potentiellement plus dans une fenêtre plus courte d'étude.

Dès lors, il serait intéressant de chercher d'autres modèles qui pourraient permettre d'obtenir des résultats potentiellement meilleurs.

## V. Critique

Principalement, j'observe une très grande différence entre les résultats annoncés et ceux que je trouve. Une explication pourrait provenir de la machine qui a fait les calculs qui provoque une certaine incertitude.

Aussi, les excellents résultats sur des logs labélisés ne sont pas réalistes mais je remarque que LogCluster semble donner de très bons résultats préliminaires. Il faut également prendre en compte le temps de calcul, difficile à comparer car certaines méthodes utilisent la carte graphique, d'autres tous les cœurs du CPU et les « plus mauvaises » n'utilisent qu'un cœur. Il y a donc une grande marge d'optimisation possible.

## VI. Conclusion

On remarque une tendance générale à l'amélioration du score F1 lorsque des labels sont utilisés et plus la méthode est récente. En revanche, il est important de noter que l'objectif des algorithmes non supervisés n'est pas de détecter des anomalies en tant que tel mais plutôt de prédire des séquences de données. En effet, si les séquences ne sont pas connues, il ne s'agit uniquement que de séquences sans sens. C'est pourquoi DeepCase s'est intéressé à réduire la charge de travail des opérateurs en indiquant dans un premier temps les séquences d'éléments à labeliser par un opérateur puis détecter les anomalies automatiquement sur ce postulat.

Ainsi, les résultats sont à prendre avec des pincettes si la question est de détecter les anomalies sur des logs. Il s'agit donc d'une différence majeure entre les algorithmes supervisés et non-supervisés mais les deux nécessiteront la connaissance, au moins partielle, des anomalies pour pouvoir les détecter et éventuellement les prévoir.

C'est sur cette dernière tâche que de nombreuses méthodes se rejoignent, la prédiction reste l'objectif principal et les méthodes non-supervisées parviennent à le faire avec des résultats hétérogènes mais la connaissance des labels sera tout de même nécessaire à un moment donné. De l'autre côté, les méthodes supervisées ont des résultats excellents et homogènes et parviennent à détecter et prédire les anomalies en tenant compte des labels, donnant la connaissance nécessaire à détecter ces anomalies.

Enfin, ce sujet est complexe et des entreprises comme Splunk (groupe Cisco) parvient à faire ce travail professionnellement mais gardent leur secret fermement. Il s'agit donc de réussir à détecter les anomalies sans la connaissance des labels, ce qu'aucune méthode ne peut faire.

## VII. Références

- [1] : Kenji Yamanishi and Yuko Maruyama. 2005. « Dynamic syslog mining for network failure monitoring ». In *KDD*. ACM, 499–508.
- [2] : Widodo A., Yang B.-S., “Support vector machine in machine condition monitoring and fault diagnosis”, *Mech Syst Signal Process*, 21 (6) (2007), pp. 2560–2574
- [3] : Xiao Han, Shuhan Yuan, Mohamed Trabelsi. “LogGPT: Log Anomaly Detection via GPT”. ArXiv 2309.14482, 2023.
- [4] : Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, Michael R. Lyu. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2023.
- [5] : Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, Michael R. Lyu. « Tools and Benchmarks for Automated Log Parsing ». *ICSE*, 2019.
- [6] : P. He, J. Zhu, Z. Zheng and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 2017, pp. 33-40, doi: 10.1109/ICWS.2017.13.
- [7] : S. Yu, P. He, N. Chen and Y. Wu, "Brain: Log Parsing With Bidirectional Parallel Tree," in *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3224-3237, Sept.–Oct. 2023, doi: 10.1109/TSC.2023.3270566.
- [8] : Wei Xu, Ling Huang, Armando Fox, David A. Patterson, Michael I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.
- [9] : J. Lou, Q. Fu, S. Yang, Y Xu, and J. Li. Mining invariants from console logs for system problem detection. In *ATC'10: Proc. of the USENIX Annual Technical Conference*, 2010.
- [10] : Q. Lin, H. Zhang, J.G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *ICSE'16: Proc. of the 38th International Conference on Software Engineering*, 2016.
- [11] : Shilin He, Jieming Zhu, Pinjia He, Michael R. Lyu. Experience Report: System Log Analysis for Anomaly Detection, *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2016.
- [12] : F. T. Liu, K. M. Ting and Z. -H. Zhou, "Isolation Forest," *2008 Eighth IEEE International Conference on Data Mining*, Pisa, Italy, 2008, pp. 413-422, doi: 10.1109/ICDM.2008.17.
- [13] : LogAnomaly: unsupervised detection of sequential and quantitative anomalies in unstructured logs by Weibin Meng, Ying Liu, Yichen Zhu et al. [Tsinghua University], 2019.
- [14] : Anomaly Detection using Autoencoders in High Performance Computing Systems, by Andrea Borghesi, Andrea Bartolini, Michele Lombardi, Michela Milano, Luca Benini, 2019.

[15] : A. Farzad, T.A. Gulliver, "Unsupervised log message anomaly detection", *ICT Express*, 6 (3) (2020), pp. 229-237

[16] : Du, M., Li, F., Zheng, G., & Srikumar, V. (2017). Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)* (pp. 1285-1298).

[17] : van Ede, T., Aghakhani, H., Spahn, N., Bortolameotti, R., Cova, M., Continella, A., van Steen, M., Peter, A., Kruegel, C. & Vigna, G. (2022, May). DeepCASE: Semi-Supervised Contextual Analysis of Security Events. In *2022 Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[18] : L. Yang *et al.*, "Semi-Supervised Log-Based Anomaly Detection via Probabilistic Label Estimation," *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, ES, 2021, pp. 1448-1460, doi: 10.1109/ICSE43902.2021.00130.

[19] : J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP. ACL*, 2014, pp. 1532-1543.

[20] : Failure Prediction in IBM BlueGene/L Event Logs, by Yinglung Liang, Yanyong Zhang, Hui Xiong, Ramendra Sahoo. 2007.

[21] : Estimating the Support of a High-Dimensional Distribution, by John Platt, Bernhard Schölkopf, John Shawe-Taylor, Alex J. Smola, Robert C. Williamson, 2001.

[22] : Detecting anomaly in big data system logs using convolutional neural network by Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang, 2018

[23] : Failure Diagnosis Using Decision Trees, by Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, Eric Brewer, 2004.

[24] : X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, et al., "Robust log-based anomaly detection on unstable log data," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 807-817, 2019.

[25] : A. Joulin, E. Grave, P. Bojanowski, T. Mikolov, *Bag of Tricks for Efficient Text Classification*, 2017.

[26] : H. Guo, S. Yuan, and X. Wu, "Logbert: Log anomaly detection via bert," arXiv preprint arXiv:2103.04475, 2021.

[27] : Xiao Han, Shuhan Yuan, Mohamed Trabelsi. "LogGPT: Log Anomaly Detection via GPT". ArXiv 2309.14482, 2023.

[28] : A. Brando and P. Georgieva, "Log Files Analysis For Network Intrusion Detection," *2020 IEEE 10th International Conference on Intelligent Systems (IS)*, Varna, Bulgaria, 2020, pp. 328-333, doi: 10.1109/IS48319.2020.9199976.

Raphaël Michon