RÉPUBLIQUE DU CAMEROUN
****
PAIX - TRAVAIL - PATRIE
****
UNIVERSITÉ DE YAOUNDÉ I
****
FACULTÉ DES SCIENCES
****
DÉPARTEMENT D'INFORMATIQUE
****

REPUBLIC OF CAMEROON
****
PEACE - WORK - FATHERLAND
****
UNIVERSITY OF YAOUNDE I
****
FACULTY OF SCIENCE
****
DEPARTMENT OF COMPUTER SCIENCE
****

Mémoire de fin de cycle

En vue de l'obtention du :
Diplôme de Master en Informatique

Option :
Apprentissage Automatique

Thème

K-MEANS RÉSILIENT AUX FAUTES BYZANTINES

Présenté par :
M. FOGANG FOKOA Kevin Jeff

Matricule :
14B2035

Sous la direction de :
Dr. MELATAGIA YONTA Paulin
Chargé de Cours, Université de Yaoundé I

Année universitaire :
2020/2021

♣ Dédicaces ♣

À ma mère chérie :

Magni Rose

Et mes sœurs bien aimées :

Makiela Fokoa Gwladys Laure

Mawe Fokoa Arlette

Mache Fokoa Lynce Vanessa

Ngume Fokoa Sonia Lucynda

♣ Remerciements ♣

Je tiens à témoigner toute ma gratitude aux personnes qui ont œuvré à la réalisation de ce mémoire de fin de cycle.

En premier lieu, je remercie le directeur de ce mémoire, **Dr. MELATAGIA YONTA Paulin**, enseignant chercheur au Département d'Informatique à l'Université de Yaoundé I, pour la confiance qu'il m'a accordé en acceptant d'encadrer ce travail de master, ainsi que pour ses conseils avisés et précieux, son aide et le temps qu'il m'a consacré.

Je tiens également à remercier **Dr. MESSI NGULE Thomas** et **Dr. HAMZA Adamou**, tous deux enseignants au Département d'Informatique de l'Université de Yaoundé I, pour leurs précieux conseils et aides qui m'ont permis d'améliorer la rédaction de ce mémoire tant sur le fond que sur la forme.

Je remercie tous les membres du jury qui ont daigné laisser leurs multiples occupations afin d'examiner ce travail. Leurs critiques et suggestions contribueront certainement à rehausser la valeur scientifique de ce travail.

Je remercie ma très chère mère, **MAGNI Rose** qui a toujours été là pour moi. Ma grande sœur **MAKIELA FOKOA Gwladys Laure** pour les multiples lectures et corrections qu'elle a apporté, ainsi que le partage de ses connaissances et expériences de ce milieu qu'est la recherche.

Enfin, je voudrais exprimer ma reconnaissance envers mes amis qui m'ont apporté leur soutien moral et intellectuel tout au long de ma démarche. Une pensée à **EDJO'O EBA Landry Steve** pour les remarques et corrections apportées.

♣ Résumé ♣

Ce mémoire s'attaque au problème de K-means distribué dans un cadre byzantin où nous avons P machines qui calculent K centroïdes à chaque itération. Parmi ces P machines, une portion ϵ ($\epsilon < 1/2$) d'entre elles est byzantine, et cette portion byzantine aura tendance à calculer K centroïdes erronés, ce qui, dans la plupart des cas, faussera l'algorithme K-means distribué. Pour corriger ces erreurs byzantines, nous utilisons comme règle d'agrégation de centroïdes la règle FABA, qui est une règle d'agrégation conçue pour agréger les vecteurs gradients calculés dans un environnement distribué byzantin. Ce mémoire ne mène qu'une étude expérimentale sur K-means couplé avec FABA. Sans toutefois apporter une preuve formelle, on constatera que le mélange de ces deux algorithmes corrige bien les erreurs byzantines jusqu'à un taux de 50% et nous permet d'obtenir des clusters assez proches des originaux.

Mots clés : K-means, Système distribué, Byzantin, Descente de gradient.

♣ Abstract ♣

This thesis is based on the problem of distributed K-means in a Byzantine framework where we have P machines which compute K centroids at each iteration. Among these P machines, a ϵ-fraction ($\epsilon < 1/2$) of them is Byzantine, and this Byzantine fraction has a tendency to compute K erroneous centroids, which, in most cases, will distort the distributed K-means algorithm. Therefore, to correct such Byzantine errors, we used a centroids aggregation rule ; the FABA rule, which is an aggregation rule designed to aggregate the calculated gradient vectors in a distributed Byzantine environment. This thesis only conducts an experimental study on K-means coupled with FABA. Without providing any formal proof, we came to notice that the mixture of these two algorithms corrects the byzantine errors well up to a rate of 50% and allows us to obtain clusters quite close to the originals.

Keywords : K-means, Distributed system, Byzantine, Gradient descent.

♣ Table des matières ♣

♣ Abréviations ♣

IA : Intelligence Artificiel
AA : Apprentissage Automatique ou Apprentissage Artificiel
DG : Descente de Gradient
DGS : Descente de Gradient Stochastique
BSP : Bulk Synchronous Parallel
SSP : Stale Synchronous Parallel
ASP : Approximate Synchronous Parallel
IQR : Interquartile Range
SSQ : Sum of Squares
DGP : Descente de Gradient Projetée
MPI : Message Passage Interface
NLP : Natural Language Processing
SPMD : Single Program Multiple Data
FABA : Fast Aggregation against Byzantine Attacks

♣ Liste des tableaux ♣

♣ Table des figures ♣

♣ Liste des Algorithmes ♣

CHAPITRE 1

INTRODUCTION GÉNÉRALE

La demande en IA et en AA a considérablement augmenté au cours des quinze dernières années. Cette croissance fulgurante a été alimentée par des progrès scientifiques et techniques aussi bien qu'en AA qu'à l'évolution du hardware et les capacités de stockage et collecte des données. Cependant, une quantité importante de données est nécessaire pour que certains algorithmes d'AA puissent être performants et efficaces afin d'obtenir de meilleurs résultats pour des applications complexes(traitement de la parole, traitement d'images, traitement de textes, etc.). De ce fait, malgré la puissance de calcul machine à disposition de nos jours, pour traiter ces grandes quantités de données, l'on a besoin davantage de puissance de calcul, et pour en obtenir, il est nécessaire de répartir la charge de travail sur plusieurs machines, d'où la nécessité de passer du centralisé au distribué.

1.1 Apprentissage automatique distribué

Un système distribué[3] est un ensemble de processeurs qui ne partagent ni mémoire ni horloge. Dans ce type de système, chaque nœud (encore appelé processus, machine ou travailleur) a sa propre mémoire locale et les nœuds communiquent entre eux via divers réseaux, tels que des bus et l'internet. Généralement, l'on répartit la charge de travail soit parce que les données sont naturellement réparties, soit parce que les données sont partitionnées entre les machines pour paralléliser le calcul.

La résolution d'un problème d'AA de manière distribuée implique une étude qui se fait en trois niveaux[1] dépendants les uns des autres : apprentissage automatique, parallélisme et topologie.

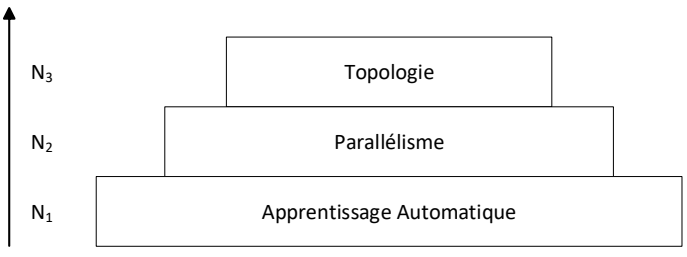


FIGURE 1.1 – Niveaux d'architecture pour résoudre un problème en AA distribué.

Apprentissage Automatique. Ici, on sélectionne l'algorithme ou modèle à utiliser. Cette sélection repose sur la nature et le type de problème que l'on essaie de résoudre. Cette partie est très importante car elle impactera sur le choix et le coût de l'implémentation distribuée de l'algorithme aux niveaux supérieurs.

Parallélisme. Après être sortie de (N1) avec un algorithme, on doit maintenant choisir quoi distribuer : ou on distribue les données, ou on distribue l'algorithme, ou alors on fait les deux. La Figure 1.2 illustre cela.

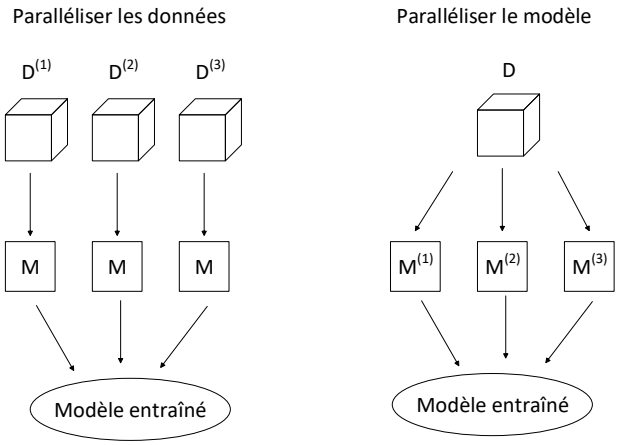


FIGURE 1.2 – Méthodes de paraléllisme en AA distibué.[1]

Topologie. On en distingue deux types : physique et logique. Dans le *physique :* il s'agit ici de choisir la structure ou l'architecture physique définissant les liaisons et une hiérarchie éventuelle entre les nœuds du système. Comme topologie physique, on peut citer la topologie en bus, anneau, arbre, client-serveur et peer-to-peer. La Figure 1.3 en illustre quelques unes. Dans le type *logique :* elle définit la manière dont les nœuds vont communiquer dans le réseau[3].

Certes, un système distribué permet de réduire le coût computationnel mais il y rajoute un coût supplémentaire appelé coût de communication. Par conséquent, un bon algorithme distribué devrait avoir un bon coût computationnel et un bon coût de communication. Pour ce faire, il existe des modèles de programmation parallèle qui tentent de prendre en compte ces deux coûts. Parmi ces modèles, on peut citer entre autres : BSP[4], SSP, ASP.

BSP. Proposé par L. Valiant[4], BSP est un méta-modèle de calcul parallèle qui n'est pas lié à un modèle de programmation ou à un matériel spécifique. Après avoir passé en entrée les données à BSP, ce dernier résume la parallélisation des programmes en trois étapes : *calcul*, *routeur* et *synchronisation*.

Calcul. Ici, nous avons un nombre fixe de composants ou unité de calculs qui exécute le programme de manière synchronisée et indépendante. Indépendant dans le sens où chaque composant effectue les calculs sur la partie des données qui lui a été assignée et ne peut qu'accéder à sa mémoire locale (d'où le nom de mémoire non-partagée ou distribuée).

Routeur. Il s'agit ici de la phase de communication où, les composants communiquent entre eux par envoie direct de messages de taille fixe.

Synchronisation : Encore appelée *Synchronisation de barrières*, elle permet d'éviter les blocages et d'assurer la cohérence des calculs. Elle marque également la fin d'une super-étape. Une super-étape est la combinaison d'étapes de *calcul*, *routeur* et d'arrivées de messages en provenance d'autres composants (barrières.), s'exécutant à intervalles de temps réguliers.

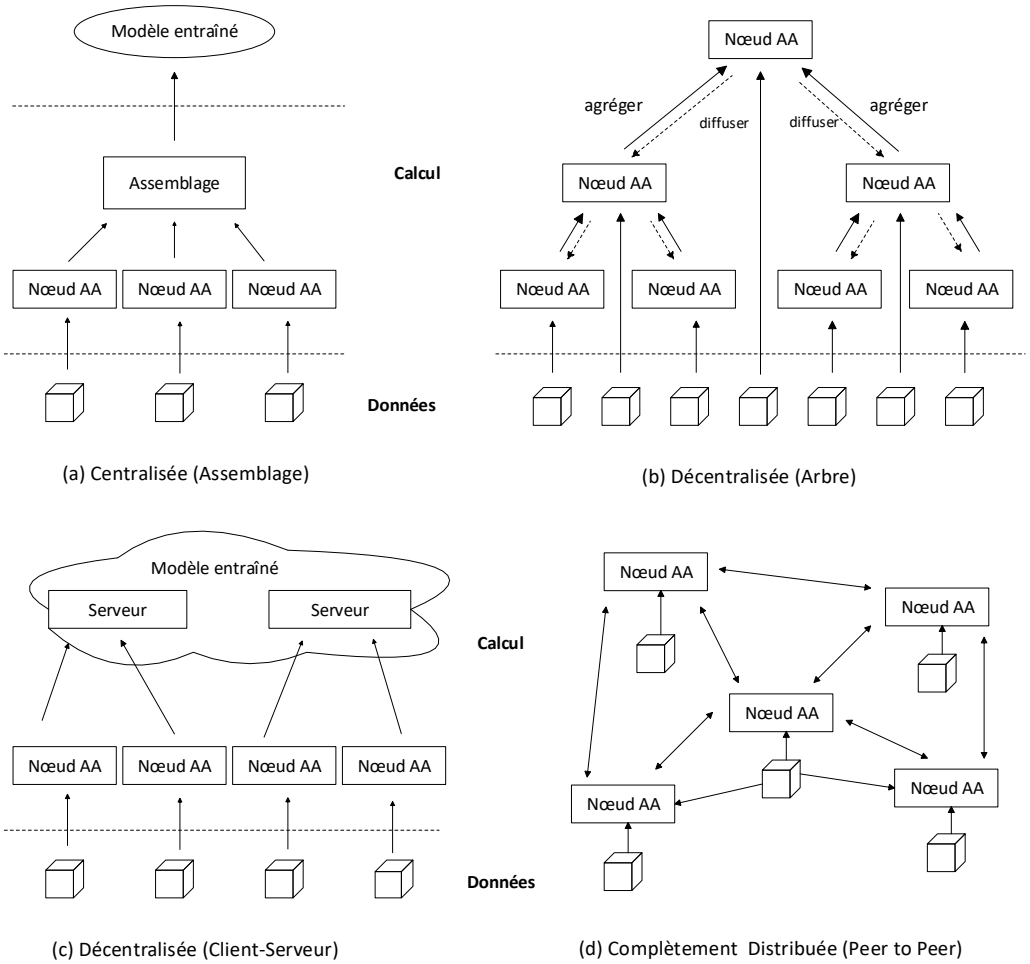


FIGURE 1.3 – Topologies en AA distribué.[1]

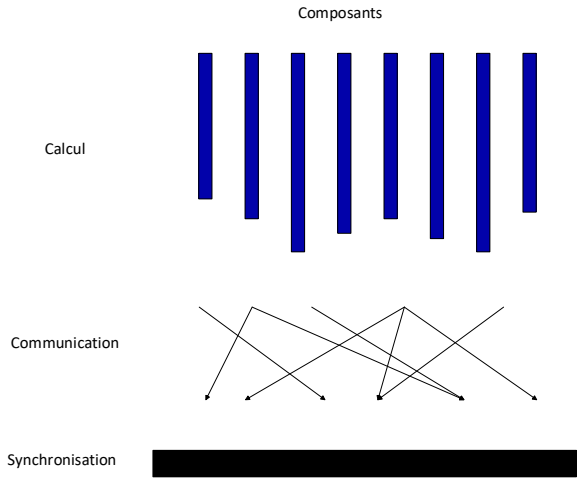


FIGURE 1.4 – Modèle BSP

Si l'AA distribué offre des bonnes performances en un temps beaucoup plus rapide, cela est dû à son paradigme mais aussi à la robustesse de ces algorithmes. Et concevoir des algorithmes robustes signifie concevoir des algorithmes qui, tout en étant efficaces, sont scalables et tolérants aux fautes. Dans ce mémoire, nous ne nous

intéressons qu'aux algorithmes tolérants aux fautes, et plus particulièrement à un type de fautes qui ne survient que dans des environnements distribués : les fautes byzantines.

1.2 Apprentissage automatique résilient

Résilience. Ce terme a plusieurs définitions selon le domaine dans lequel nous nous trouvons mais garde son essence. En effet, selon Larousse [1] en physique, la résilience est une caractéristique mécanique définissant la résistance aux chocs d'un matériau ; en psychologie, ce terme renvoie aux aptitudes qu'a un individu à se construire et à vivre de manière satisfaisante en dépit de circonstances tragiques. Selon Larousse toujours, la résilience en écologie est la capacité d'un écosystème, d'un biotope ou d'un groupe d'individus (population, espèce) à se rétablir après une perturbation extérieure (catastrophe naturelle, etc) ; cette définition de la résilience dans l'écologie s'illustre à merveille avec l'AA dans [5] où S. Saravi et al. utilisent les méthodes d'AA pour améliorer la résilience et la capacité des individus, des communautés et des États à s'adapter aux inondations. Et enfin, en informatique, la résilience c'est la capacité qu'a un système à continuer de fonctionner en cas de panne d'un ou plusieurs de ses éléments .

On peut donc définir la *résilience des algorithmes* comme la capacité qu'auront les algorithmes à continuer de fonctionner en cas d'erreur, en cas d'attaque et de supporter la montée en charge. C'est pour répondre à ces besoins que M. A. Heroux[6] donne les quatre modèles de programmation permettant de rendre les algorithmes résilients.

Apprentissage automatique résilient. Dû aux progrès et l'efficacité qu'offre l'AA, nombreuses sont des applications sensibles qui reposent désormais sur des systèmes d'AA. On peut citer entre autres : la surveillance et la détection d'intrusion en sécurité, la détection d'anomalies en médecine, les véhicules autonomes. Ces systèmes sont vulnérables aux attaques [2] d'adversaires [3] et cela à toutes les phases de l'apprentissage. La résilience des algorithmes d'AA est devenue un axe de recherche de plus en plus étudié et mieux compris. Les algorithmes/modèles/systèmes d'AA sont principalement susceptibles à trois types d'attaques[7] : les attaques exploratoires, les attaques par empoisonnement et les attaques d'évasion.

Les attaques exploratoires : elles ont pour but de découvrir les informations concernant le modèle qui est entrain d'être (ou qui va être) exécuté. Les informations que l'adversaire tente de découvrir sont par exemple l'état du modèle sous-jacent, l'algorithme et les données utilisés par le système. Cette phase exploratoire déterminera quelles techniques d'attaques malveillantes l'adversaire entreprendra. Les attaques exploratoires ne sont pas malveillantes.

1. https ://www.larousse.fr/dictionnaires/francais/résilience

2. Une attaque est l'exploitation d'une faille d'un système informatique (système d'exploitation, logiciel ou bien même de l'utilisateur) à des fins non connues par l'exploitant du systèmes et généralement préjudiciables. https ://web.maths.unsw.edu.au/ lafaye/CCM/attaques/attaques.htm

3. Un adversaire est une entité qui n'est pas autorisée à accéder aux informations ni à les modifier, ou qui s'efforce de contourner les protections accordées aux informations. https ://csrc.nist.gov/glossary/term/adversary

Les attaques par empoisonnement : elles tentent d'influencer les données d'entraînement en y injectant des données qui n'ont rien à voir avec la distribution des données d'origine pour influencer le résultat de l'apprentissage. On retrouve naturellement cette attaque dans les données sous forme d'outliers [4]. La recherche sur l'apprentissage en présence des outliers avait été abandonnée dans les années 60-70 et a regagné de l'intérêt ces dernières années. C'est dans ce regain d'intérêt que J. Steinhardt [8] établit une théorie d'information sur la notion de résilience et se base dessus pour concevoir des algorithmes de calcul de moyenne basés sur des valeurs propres et la dualité des problèmes de point de selle.

Les attaques d'évasion : contrairement aux attaques par empoisonnement qui se déroulent pendant la phase d'entraînement du modèle, les attaques d'évasion se déroulent pendant la phase d'inférence. Les entrées malveillantes sont soigneusement conçues pour qu'elles puissent ressembler à leur copie non altérée pour un humain mais qui perturbent complètement le système d'AA.

Byzantin. Les systèmes (y compris les systèmes d'AA) qui s'exécutent dans des environnements distribués sont soumis à un type d'erreur ou faute appelé byzantin[9]. *Une erreur byzantine :* est une erreur qui survient lorsqu'une fraction de nœuds d'un système distribué peut envoyer des données corrompues ou malveillantes et, peut tomber en panne de manière imprévisible et arbitraire. Ces nœuds sont appelés nœuds byzantins.

Environnement distribué byzantin. Dans toute la suite de ce mémoire, nous considérons un environnement distribué fonctionnant sur une architecture client-serveur ou travailleuses-coordonatrice (les travailleuses ne peuvent pas communiquer entre elles mais peuvent communiquer avec la coordonatrice) suivant le modèle BSP dans lequel nous avons :

- Une machine coordonatrice et P machines travailleuses ayant pour identifiant respectif $0, 1, 2, \ldots, P$.
- un sous-ensemble inconnu $\mathcal{G}$ de γP machines appelé ensemble de bonnes machines où γ est inconnu et appartient à $[0, 1]$;
- et un sous-ensemble $\mathcal{B}$ (aussi inconnu) de $(1-\gamma)P$ machines appelé ensemble de mauvaises machines (machines byzantines). Ces machines peuvent se comporter de manière défavorable.

En posant $\epsilon = (1 - \gamma)$, à la suite de ce mémoire, considéra $\epsilon < \frac{1}{2}$.

La conception d'algorithmes distribués d'AA résilients aux machines byzantines anime la recherche scientifique actuelle et c'est sur ce thème que porte ce mémoire.

1.3 Problématique

Les algorithmes d'apprentissage automatique peuvent être divisés en trois catégories [1] : le type, le but et la méthode d'apprentissage. Selon leur type, on peut encore les classer en trois catégories à savoir : l'apprentissage supervisé, l'apprentissage non-supervisé et l'apprentissage par renforcement. Dans ce mémoire, nous nous

4. Un outlier est un point de données qui diffère considérablement des autres points. Il peut être du à la variabilité de la mesure ou à une erreur expérimentale.

intéressons plus aux algorithmes d'apprentissage automatique non-supervisé ; particulièrement à l'algorithme de clustering : K-means.

K-means clustering est une méthode permettant de partitionner des objets en K groupes d'objets similaires. C'est l'algorithme de clustering le plus utilisé et le plus populaire aussi bien en analyse de données qu'en fouille de données. Il a de nombreuses applications parmi lesquelles la récupération de l'information dans le traitement de texte et la segmentation d'image en traitement d'images.

Faire du K-means sur un ensemble de données de grande taille nécessite beaucoup de temps et de puissance de calcul. Pour pallier ce problème, l'on peut distribuer l'algorithme K-means. Mais comme tout algorithme s'exécutant dans un environnement distribué, K-means sera sous la menace des machines byzantines.

Ce mémoire s'attaque au problème fondamental qui est de savoir ***comment rendre l'algorithme K-means distribué résilient aux fautes byzantines ?***

Afin de proposer une solution à ce problème, nous nous sommes orientés vers les algorithmes qui permettent de rendre les méthodes de descente de gradient résilients aux machines byzantines. C'est ainsi que nous y avons sélectionné l'algorithme FABA car il est facile à implémenter et est le plus performant [5] de tous [2].

1.4 Plan du mémoire

Nombreux sont les algorithmes de descente de gradient distribués résilients aux fautes byzantines dans la littérature scientifique. Tous consistent à concevoir des règles d'agrégation à partir de certaines hypothèses et ayant des propriétés de convergence et de robustesse ; et c'est par ces règles d'agrégation que ces algorithmes diffèrent. Dans le Chapitre 2, nous étudions les propriétés de convergence de la méthode de descente de gradient à la Section 2.2 après avoir donné quelques définitions de base à la Section 2.1 ; nous présentons ensuite aux Sections 2.3 et 2.4 quelques algorithmes de descente de gradient distribués résilients, tandis qu'à la section 2.5, nous parlons des algorithmes K-means, K-means distribué et établissons le lien qu'il y a entre K-means et la méthode de descente de gradient. Dans le Chapitre 3, la Section 3.1 décrit K-means byzantin, c'est ici que nous verrons comment les erreurs byzantines peuvent influencer un algorithme ; Dans la Section 3.2 nous nous focalisons sur K-means résilient aux byzantines qui n'est rien d'autre que le résultat de la combinaison de FABA et K-means byzantin. Dans la Section 3.3, nous donnons l'environnement matériel et logiciel sur lesquels nos expérimentations ont été faites, ainsi que les résultats obtenus. Et enfin, à la Section 3.4 nous discutons des propriétés théoriques de notre modèle.

5. FABA est le meilleur algorithme de DG distribué à l'heure où nous entamons ce mémoire (2020).

CHAPITRE 2

GÉNÉRALITÉS ET ÉTAT DE L'ART SUR LES ALGORITHMES BYZANTINS

La performance des algorithmes d'apprentissage automatique dépend de la méthode d'apprentissage et, la méthode la plus utilisée est la méthode de descente du gradient (et ses variantes), qui est une méthode d'optimisation mathématiques.

2.1 Prérequis

Les définitions et notations suivantes seront utilisées tout au long de ce mémoire.

Notation 2.1.

- $\|\cdot\|$ *dénote la norme Euclidienne.*
- $[n] = \{1, 2, \ldots, n\}$.
- $f^* = f(x^*)$.
- ∇f *dénote le gradient de la fonction* f.

Définition 2.2. *Soient $\mathcal{X} \subset \mathbb{R}^d$ un ensemble convexe et f une fonction différentiable de $\mathcal{X}$ à valeur dans $\mathbb{R}$:*

1. *f est convexe si $\forall x, y \in \mathcal{X}\ f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle$.*
2. *f est σ-fortement convexe si $\exists\ \sigma \in \mathbb{R}_+^*, \forall\ x, y \in \mathcal{X},\ f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{\sigma}{2}\|x - y\|^2$.*
3. *f est L-smooth si $\exists\ L \in \mathbb{R}_+^*, \forall\ x, y \in \mathcal{X},\ \|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$.*
4. *f est G-Lipschitz continue si $\forall\ x \in \mathcal{X},\ \|\nabla f(x)\| \leq G$.*

En apprentissage automatique, et plus particulièrement en apprentissage supervisé, on dispose d'une collection de taille fixe d'échantillons (encore appelés exemples ou réalisations) $\{(x_i, y_i)\}_{i=1}^n$ où $(x_i, y_i) \in \mathbb{R}^d \times \mathbb{R}$. Ces échantillons [1] sont des réalisations de n couples de variables aléatoires $\{(X_i, Y_i)\}_{i=1}^n$ pris de manière indépendantes et identiquement distribués de même loi que (X, Y).

Le but en apprentissage supervisé est de trouver une fonction de prédictions h qui minimise une fonction (fonction objective différentiable et convexe) de perte $l\left(h(x; w), y\right)$[10]. Minimiser une fonction nécessite l'utilisation des méthodes d'optimisation.

1. Dans la collection en question, les y_i ne sont pas toujours continus. Ils peuvent aussi appartenir à un ensemble discret.

2.2 Algorithme de descente de gradient

L'algorithme de descente de gradient est une méthode d'optimisation différentiable sans contrainte de premier ordre qui permet de trouver un minimum local d'une fonction objective différentiable. Formellement, cela revient à poser le problème de minimisation d'une fonction objective différentiable f :

$$f^* = \min_{w \in \mathbb{R}^d} \{f(w)\} \tag{2.1}$$

L'algorithme de DG possède plusieurs variantes selon la manière dont est calculé le vecteur gradient. Nous pouvons citer entre autres : DG vanilla, DGS, DGS-minibatch.

Descente de gradient vanilla. Encore appelé la *descente de gradient batch*, elle consiste à calculer en fonction de w, le gradient de la fonction coût l sur tout le jeu de données. Dans ce cas, le f du problème (2.1) vaut :

$$f(w) = \frac{1}{n} \sum_{i=1}^{n} l\left(h(x_i; w), y_i\right)$$

Algorithme 1 Descente de gradient

Entrée :

η_k : taux d'apprentissage, supérieur à 0.
T : nombre d'itérations.
w_0 : valeur initiale, $\in \mathbb{R}^d$.

1. **Pour** $t \leftarrow 0$ **à** $T - 1$ **faire**
2. $\quad w_{t+1} = w_t - \eta_t \nabla f(w_t)$;

En sortie : w_T.

À la ligne 2 de cet algorithme, $\nabla f(w_t) = \frac{1}{n} \sum_{i=1}^{n} \nabla l\left(h(x_i; w), y_i\right)$. Il existe plusieurs dérivées de cet algorithme, qui diffèrent les unes des autres par la stratégie du choix du taux d'apprentissage[11].

Dans certains cas, résoudre un problème de minimisation pourrait signifier trouver une solution exacte. Cependant, dans de nombreux problèmes cela est impossible. Résoudre donc le problème d'optimisation signifie parfois trouver une solution approchée avec une précision $\varepsilon > 0$, avec ε assez petit. Donc, on aimerait que l'Algorithme 1 retourne un $f(w_T)$ tel que : $f(w_T) - f^* \leq \varepsilon$.

De nombreux travaux dans la littérature scientifique porte sur la convergence de cette méthode en fonction de différentes suppositions faites sur la fonction objective.

Théorème 2.3. *Considérons le problème 2.1. Supposons que* $\|w_0 - w^*\| \leq D$ *et* $\|\nabla f(w)\| \leq G$ *; et qu'en choisissant* η_t *de manière optimale* ($\eta_t = \frac{D}{G\sqrt{T}}$)*, on a :*

$$\frac{1}{T} \sum_{t=0}^{T-1} f(w_t) - f^* \leq \frac{GD}{\sqrt{T}} \tag{2.2}$$

Le Théorème 2.3 stipule que pour que l'Algorithme 1 atteigne une précision $\varepsilon > 0$(selon les suppositions faites), l'on a besoin de $O\left(\frac{1}{\varepsilon^2}\right)$ itérations. En effet, d'après 2.2, si on prend : $T \geq \frac{G^2D^2}{\varepsilon^2} \Rightarrow \frac{GD}{\sqrt{T}} \leq \varepsilon \Rightarrow \frac{1}{T}\sum_{t=0}^{T-1} f(w_t) - f^* \leq \frac{GD}{\sqrt{T}} \leq \varepsilon$. Donc, si on veut par exemple obtenir une précision de 0.01, nous aurons besoin d'au moins $\frac{1}{0.01^2} = 10000$ itérations.

Théorème 2.4. *(Corollaire 4.3 [12]) Considérons le problème 2.1. Supposons que f est L-smooth et $\|w_0 - w^*\| \leq D$; et qu'en choisissant η_t de manière optimale ($\eta_t = \frac{1}{L}$), on a :*

$$f(w_T) - f^* \leq \frac{LD^2}{2T} \tag{2.3}$$

Le Théorème 2.4 stipule que pour que l'Algorithme 1 atteigne une précision $\varepsilon > 0$, l'on a besoin de $O\left(\frac{1}{\varepsilon}\right)$ itérations. En effet, d'après 2.3, si on prend : $T \geq \frac{LD^2}{2\varepsilon} \Rightarrow f(w_T) - f^* \leq \frac{GD}{\sqrt{T}} \leq \varepsilon$. Donc, si on veut par exemple obtenir une précision de 0.01, nous aurons besoin d'au moins de $\frac{1}{0.01} = 100$ itérations.

Théorème 2.5. *(Corollaire 4.11 [12]) Considérons le problème 2.1. Supposons que f est L-smooth, σ-fortement convexe et $\|w_0 - w^*\| \leq D$; et qu'en choisissant η_t de manière optimale ($\eta_t = \frac{1}{L}$), on a :*

$$f(w_T) - f^* \leq \frac{L}{2}\left(1 - \frac{\sigma}{L}\right)^T D^2 \tag{2.4}$$

À partir du Théorème 2.5, pour obtenir une précision de $\varepsilon > 0$, l'on a besoin de $O\left(\ln(\frac{1}{\varepsilon})\right)$ itérations. En effet, sachant que $(1 - x) \leq \exp(-x)$, posons $(1 - \sigma/L) \leq \exp(-\sigma/L) \Rightarrow (1 - \sigma/L)^T \leq \exp(-T\sigma/L)$; d'après 2.4, si on prend : $T \geq \frac{L}{\sigma}\ln\left(\frac{LD^2}{2\varepsilon}\right) \Rightarrow f(w_T) - f^* \leq \frac{L}{2}\left(1 - \frac{\sigma}{L}\right)^T D^2 \leq \varepsilon$. Ainsi, si on veut par exemple obtenir une précision de 0.01, nous aurons besoin d'au moins $\ln\left(\frac{1}{0.01}\right) \simeq 5$ itérations

L'algorithme de descente de gradient possède des avantages et des inconvénients. Comme avantages, c'est un algorithme simple qui est facile à implémenter et qui n'est pas coûteux car il a juste besoin de calculer le vecteur gradient à chaque itération. Il est très rapide lorsque la fonction objective est bien conditionnée comme dans le Théorème 2.5. Comme inconvénients, il ne peut pas s'exécuter sur des fonctions non différentiables et peut être très lent lorsqu'il est exécuté sur des données volumineuses.

Descente de gradient stochastique. Ici, on calcule en fonction de w le gradient de la fonction coût l sur un échantillon pris uniformément dans $\{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$. Dans cette variante, le problème (2.1) est reformulée de la manière suivante :

$$f^* = \min_{w \in \mathbb{R}^d} \{f(w) := \mathbb{E}[l(h(x; w), y)|w]\} \tag{2.5}$$

où l est convexe et différentiable.

L'algorithme de descente de gradient stochastique est quasiment le même que l'Algorithme 1. La différence est au niveau de la ligne 2, sur le calcul du vecteur gradient où $\nabla f(w)$ devient $\nabla f_i(w)$ avec $f_i(w) = l\left(h(x_i; w), y_i\right)$ où $i \in [n]$. Pour plus

de détails voir [10, 13].

Algorithme 2 Descente de gradient stochastique

Entrée :

η_k : taux d'apprentissage, supérieur à 0.
T : nombre d'itérations.
w_0 : valeur initiale, $\in \mathbb{R}^d$.

1. **Pour** $t \leftarrow 0$ **à** $T-1$ **faire**
2. Soit i choisi aléatoirement dans $[n]$;
3. $w_{t+1} = w_t - \eta_t \nabla f_i(w_t)$;

En sortie : w_T.

L'algorithme de DGS vient pallier le problème de coût élevé qu'offre la DG sur des données de grande taille car il ne calcule qu'un seul gradient par itération. Malgré cet avantage, la DGS pose un problème de fluctuation lors de la convergence, ce qui peut nécessiter davantage d'itérations pour converger vers le minimum local.

Pour étudier la convergence de cette méthode en fonction de différentes suppositions faites sur la fonction objective, on pose $g_t = g(w_t) = \nabla f_i(w_t)$ où g_t est un vecteur de d variables aléatoires et un estimateur non-biaisé de $\nabla f(w_t)$ c'est-à-dire $\mathbb{E}[g(w_t)] = \nabla f(w_t)$.

Théorème 2.6. *[13] Considérons le problème 2.5. Supposons que* $\|w_0 - w^*\| \leq D$, $\forall\, t$, $\mathbb{E}[\|g_t\|^2] \leq G^2$ *; et qu'en choisissant* $\eta_t = \frac{D}{M\sqrt{T}}$*, on a :*

$$\mathbb{E}[f(\tilde{w}_T) - f^*] \leq \frac{GD}{\sqrt{T}} \qquad \text{où } \tilde{w}_T = \sum_{t=0}^{T-1} \frac{\eta_t}{\sum_{t=0}^{T-1} \eta_t} w_t \tag{2.6}$$

À partir du Théorème 2.6, pour une précision $\varepsilon > 0$, l'on a besoin de $O\left(\frac{1}{\varepsilon^2}\right)$ itérations. Même raisonnement que celui du Théorème 2.3.

Théorème 2.7. *[13] Considérons le problème 2.5. Supposons que* f *est* L*-smooth,* σ*-fortement convexe,* $\|w_0 - w^*\| \leq D$ *et* $\forall\, t$, $\mathbb{E}[\|g_t\|^2] \leq G^2$ *; en choisissant* $\eta_t = \frac{\theta}{T}$ *où* $\theta > \frac{1}{2\sigma}$*, on a :*

$$\mathbb{E}[f(w_T) - f^*] \leq \frac{L \max\{\theta^2 G^2 (2\theta\sigma - 1)^{-1}, D^2\}}{2T} \tag{2.7}$$

À partir du Théorème 2.7, pour obtenir $f(w_T) - f^* \leq \varepsilon$, l'on a besoin de $O\left(\frac{1}{\varepsilon}\right)$ itérations. Même raisonnement que celui du Théorème 2.4.

Lorsque la fonction objective est bien conditionnée, on constate que la DGS nécessite beaucoup plus d'itérations pour converger contrairement à la DG qui en nécessite beaucoup plus pour atteindre l'erreur souhaitée.

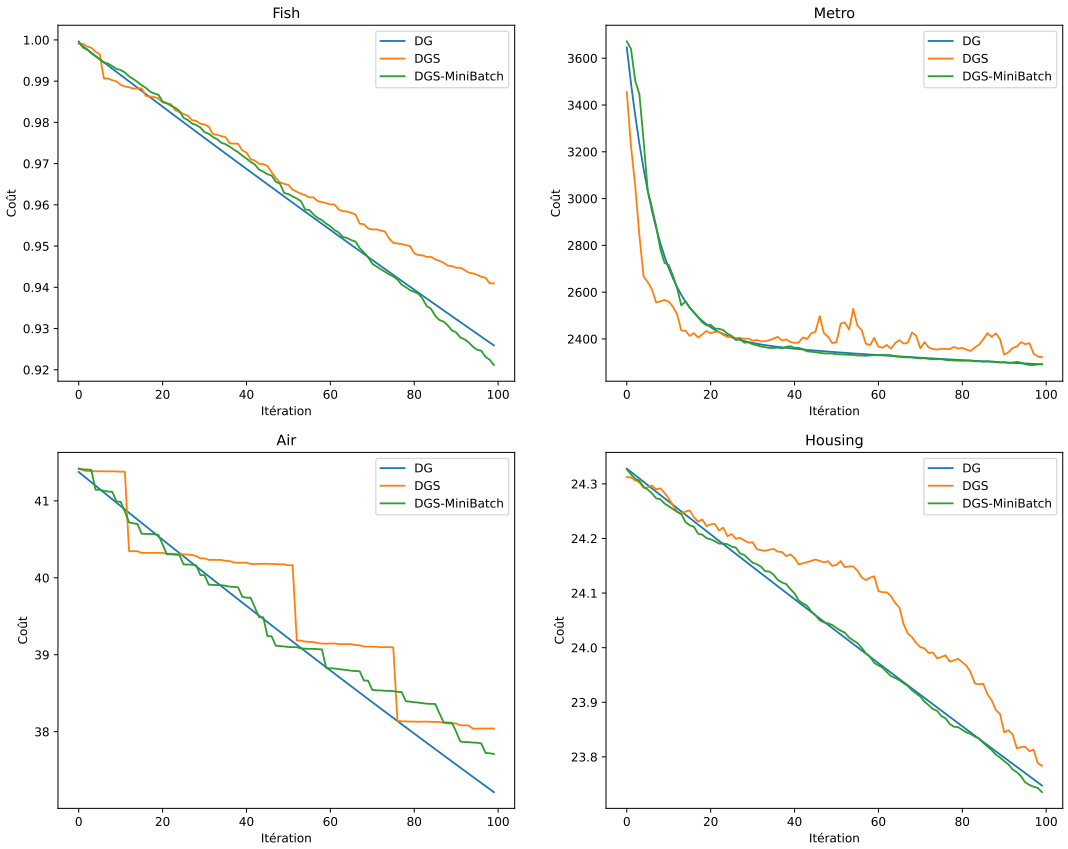


FIGURE 2.1 – Méthodes de descente de gradient appliquées à 4 jeux de données.
[Source : *https ://github.com/fokoa/linear_regression/blob/main/memoire.ipynb*, Accès : *20 Nov 21.*]

Descente de gradient stochastique mini batch. Cette méthode vient résoudre le problème de DG (lent lorsque n est très grand) et DGS (variance élevée) en calculant le gradient à partir d'une portion de données. L'algorithme est le suivant :

Algorithme 3 Descente de gradient stochastique mini batch

Entrée :

η_k : taux d'apprentissage, supérieur à 0.
T : nombre d'itérations.
w_0 : valeur initiale, $\in \mathbb{R}^d$.

1. **Pour** $t \leftarrow 0$ **à** $T-1$ **faire**
2. $\quad w_{t+1} = w_t - \frac{\eta_t}{m} \sum_{i=1}^{m} \nabla f_i(w_t)$;

En sortie : w_T.

Cet algorithme réduit la variance de DGS d'un facteur de $\frac{1}{m}$ car le gradient est calculé à partir de m ($m < n$) échantillons et le taux de convergence de l'Algorithme 3 montre que, le taux de convergence de SGD est divisé par m [10].

La Figure 2.1 illustre ces trois algorithmes. En raison des mises à jour fréquentes, la DGS fluctue beaucoup et peut nécessiter davantage d'itérations . Cela peut souvent faire pencher la descente du gradient dans d'autres directions (Fish et Air).

2.3 Algorithmes de descente de gradient résilient : Krum et FABA

L'optimisation mathématiques étant au cœur des algorithmes d'AA, il est tout naturel que l'algorithme de DG soit le plus étudié dans l'environnement distribué car dans le centralisé, il est le plus populaire, le plus étudié et le plus simple. La conception des algorithmes DG distribués qui convergent vers la bonne solution f^* en présence des machines byzantines est bien étudiée.

2.3.1 Krum

Krum[14]. C'est une règle d'agrégation non-linéaire (exécutée par la coordinatrice) basée sur la méthode de majorité et la méthode du carré de la distance. Majorité parce qu'elle calcule γP sous-ensembles pour en sélectionner par la suite celui qui a le plus petit diamètre. Carré de la distance parce que le gradient final est un vecteur qui minimise la somme des distances au carré. Associer Krum à l'algorithme de descente de gradient distribué permet de rendre ce dernier résilient aux machines byzantines lorsque $2\epsilon P + 2 < P$.

Description. P. Blanchard et al.[14] considèrent un environnement distribué défini comme dans la Section 1.2. Ils procèdent en plusieurs itérations comme suit : À chaque itération t, la coordinatrice diffuse le paramètre $w_t \in \mathbb{R}^d$ à toutes les travailleuses. Puis, chaque bonne travailleuse i calcule une estimation $g_i^t = G(w_t, \xi_i^t)$ et l'envoie à la coordinatrice, où ξ_i^t est une variable aléatoire et $g_i^t \sim G(w_t, \xi_i^t)$ avec $\mathbb{E}_{\xi_i^t}[G(w_t, \xi_i^t)] = \nabla f(w_t)$. Une travailleuse byzantine $b \in \mathcal{B}$ fournira un vecteur g_b^t qui peut être très différent du vecteur correct qui aurait dû être envoyé. Les auteurs considèrent que si la coordinatrice ne reçoit pas un vecteur g_b^t alors elle considère que b lui a envoyé un vecteur nul c'est-à-dire que $g_b^t = 0_{\mathbb{R}^d}$. La coordinatrice, après avoir reçu les P vecteurs gradients, les agrègent à l'aide de la règle d'agrégation Krum noté $F(g_1^t, g_2^t, \ldots, g_P^t)$. Une fois agrégés, la coordinatrice met à jour le paramètre comme suit :

$$w_{t+1} = w_t - \eta_t F(g_1^t, g_2^t, \ldots, g_P^t).$$

Contributions. P. Blanchard et al.[14] apportent trois contributions majeures :
(1) Ils prouvent qu'aucune approche par combinaison linéaire classique ne peut converger en présence d'une seule machine byzantine car à partir cette byzantine, on peut toujours faire en sorte que la règle d'agrégation sélectionne un vecteur quelconque U.

Lemme 2.8. *[14] Soit une règle d'agrégation linéaire F de la forme*

$$F(g_1, g_2, \ldots, g_P) = \sum_{p=1}^{P} \lambda_p g_p \ , \qquad \lambda_p \in \mathbb{R} \text{ avec } p \in [P].$$

Soit $U \in \mathbb{R}^d$ un vecteur. Une seule machine byzantine peut faire en sorte que F sélectionne toujours U. Par conséquent, une seule byzantine peut empêcher la convergence.

(2) Ils donnent une définition formelle de ce qu'est un algorithme résilient aux byzantines.

Définition 2.9. *($(\alpha, c = \epsilon P)$-Résilient Byzantin). Soit un angle α et un entier c tels que : $0 \leq \alpha < \pi/2$ et $0 \leq c \leq P$. Soit $g_1, \ldots, g_P$ des vecteurs aléatoires indépendantes et identiquement distribués dans $\mathbb{R}^d$, $g_i \sim G$, avec $\mathbb{E}[G] = g$. Soit $b_1, \ldots, b_c$ des vecteurs aléatoires dans $\mathbb{R}^d$, possiblement dépendant des $g_i's$. La règle d'agrégation F est dite (α, c)-Résilient Byzantin si pour tout $1 \leq j_1 < \cdots < j_c \leq P$, le vecteur :*

$$F = F(g_1, \ldots, \underbrace{b_1}_{j_1}, \ldots, \underbrace{b_c}_{j_c}, \ldots, g_P)$$

satisfait :

1. $\langle \mathbb{E}[F], g \rangle \geq (1 - \sin\alpha) \cdot \|g\|^2 > 0$.
2. *pour $r = 2, 3, 4$, $\mathbb{E}\left[\|F\|^r\right]$ admet une borne supérieure par une combinaison linéaire de termes $\mathbb{E}[\|G\|^{r_1}] \ldots \mathbb{E}[\|G\|^{r_{P-1}}]$ avec $r_1 + \cdots + r_{P-1} = r$.*

(3) Ils donnent une règle d'agrégation résiliente nommée Krum de complexité $O(P^2 \cdot d)$[14] et prouvent que celle-ci, couplée avec la descente de gradient convergent (à condition que $2c + 2 < P$) toujours malgré les c byzantines.

Algorithme 4 Krum

Entrée :

$g = \{g_1, \ldots, g_P\}$: g_i est le gradient calculé par la machine i avec $i \in [P]$.
P : nombre de machines.
c : nombre de machines byzantines.

1. **Pour** chaque g_i dans g **faire**
2. $\quad g' \leftarrow P - c - 2$ plus proches vecteur de g_i ;
3. $\quad d \leftarrow 0$;
4. $\quad\quad$ **Pour** chaque g_j dans g' **faire**
5. $\quad\quad\quad d \leftarrow d + \|g_i - g_j\|^2$;
6. $\quad\quad s(i) \leftarrow d$;

7. $i_* = \arg\min_i(s(i))$;

En sortie : g_{i_*}

Le défaut de *Krum* est qu'il perd beaucoup d'informations car parmi les potentiels bons γP vecteurs gradients, il n'en sélectionne qu'un seul (g_{i_*}). Cela cause par la suite des fluctuations au cours des itérations.

2.3.2 FABA

FABA est un algorithme rapide d'agrégation de gradients résilient aux machines byzantines qui élimine les gradients aberrants pour en obtenir que ceux qui sont proches des vrais gradients. Il a été conçu par Q. Xia et al.[2] et est une amélioration de Krum comme indiqué ci-dessous.

Description. Q. Xia et al.[2] se placent dans un environnement distribué byzantin défini comme dans la Section 1.2 où chaque machine dispose d'une copie conforme du modèle à entraîner. Puis ils procèdent en itérations structurées comme suit : À chaque itération t, chaque machine i entraîne le modèle sur le jeu de données qui lui a été assigné, calcule son gradient g_i puis l'envoie à la coordinatrice. La coordinatrice agrège ensuite les gradients g_i puis envoie le paramètre mis-à-jour à toutes les machines. La mise à jour se fait comme suit

$$w_{t+1} = w_t - \eta_t F(\bar{g}_1, \bar{g}_2, \ldots, \bar{g}_P)$$

où $F(\cdot)$ est la fonction d'agrégation des gradients. Les $\bar{g}_i$ sont définit comme suit

$$(\bar{g}_i)_j \leftarrow \begin{cases} (g_i)_j & \text{si la } j\text{-ième coordonnée de } g_i \text{ est correcte} \\ * & \text{sinon} \end{cases}$$

Contributions. Q. Xia et al.[2] n'apportent qu'une seule contribution. Il s'agit de FABA, qui est la meilleure règle d'agrégation connue [2] pouvant supporter jusqu'à ϵP byzantines. En effet, FABA est une sorte de moyenne élaguée où, à chaque itération t, on calcule la distance entre chaque g_i et g_0 (la moyenne de tous les gradients reçu par la coordinatrice). La Figure 2.2 l'illustre. Ensuite, on supprime le g_i qui a la plus grande distance avec g_0. On répète ce processus ϵP fois. Donc, on supprime ϵP gradients prétendument byzantins à chaque itération. FABA repose sur les mêmes hypothèses que celles de Krum et vient corriger le défaut de ce dernier. En effet, FABA est plus stable et converge[2] beaucoup plus rapidement que Krum qui fluctue beaucoup et converge lentement. Ceci est dû au fait que FABA conserve $(1-\epsilon)P$ gradients par itération tandis que Krum ne conserve qu'un seul gradient par itération (presque tous les bons gradients sont éliminés).

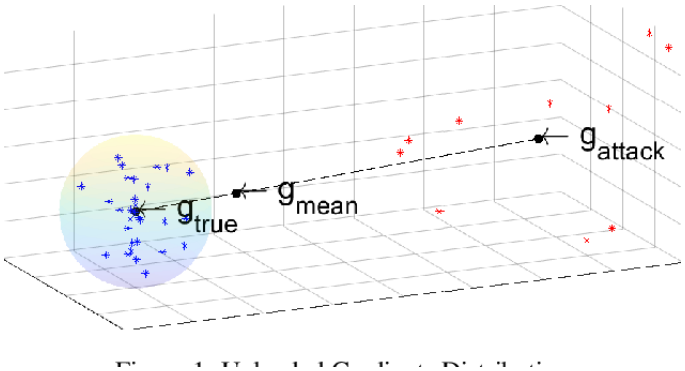


FIGURE 2.2 – Elimination des gradients supposés byzantins.[2]

2. FABA est le meilleur algorithme de DG distribué sur le plan pratique à l'heure où nous entamons ce mémoire (2020)

Algorithme 5 FABA

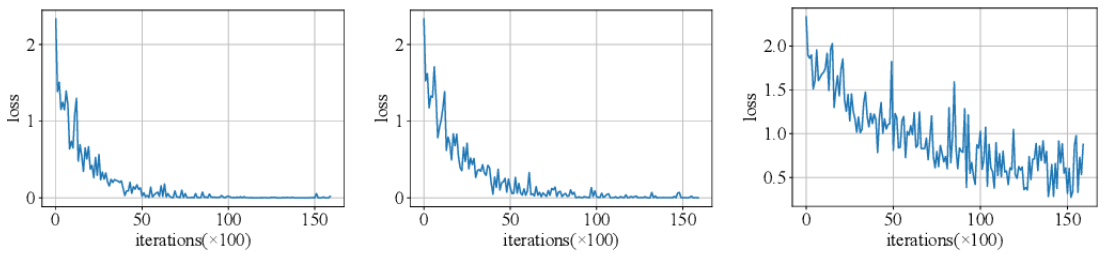
Entrée :

$g = \{g_1, \ldots, g_P\}$: les gradients estimés par la machine i avec $i \in [P]$.
ϵP : la proportion de machines byzantines

1. $j \leftarrow 1$;
2. **si** $j < \epsilon P$, continuer, **sinon** aller à l'étape 6 ;
3. Calculer la moyenne de g comme g_0 ;
4. **Pour** chaque g_i dans g, calculer la différence entre g_i et g_0.
   Supprimer de g celui qui à la plus grande difference ;
5. $j \leftarrow j + 1$ et aller à l'étape 2 ;
6. $\bar{g} \leftarrow$ la moyenne des g_i restant dans g ;

En sortie : $\bar{g}$

De la ligne 3 de cet algorithme, FABA est sensible aux valeurs aberrantes car la moyenne n'est pas robuste. Pour améliorer FABA, Noutcha[15] a proposé à la place de la moyenne arithmétique des méthodes statistique classique de suppression de valeurs aberrantes comme IQR et Z-Score.



(a) No Byzantine (b) FABA (c) Krum

FIGURE 2.3 – Sur un environnement contenant 30% de byzantines, FABA vs Krum. FABA ne fluctue pas beaucoup et converge assez rapidement contrairement à Krum qui fluctue beaucoup et converge lentement [2].

2.4 Algorithme de descente du gradient résilient de Dan Alistarh et al.

ByzantineSGD conçu par D. Alistarh et al.[16], est un algorithme d'agrégation de descente de gradient distribué beaucoup plus complexe à analyser que ceux de P. Blanchard et al.[14] et Q. Xia et al.[2].

Contrairement aux deux algorithmes mentionnés dans la Section 2.3 qui minimisaient une fonction objective sans contrainte, dans [16], il s'agit de minimiser une fonction objective avec contrainte. Pour se faire, les auteurs utilisent une méthode d'optimisation avec contrainte de premier ordre appelé *descente de gradient projetée*[17]. Le problème de minimisation qu'ils essayent de résoudre est le suivant :

$$\min_{\|w - w_0\| < D} \left\{ f(w) = \mathbb{E}_{s \sim \mathcal{D}} \left[f_s(w) \right] \right\} \tag{2.8}$$

où $\mathcal{D}$ est une distribution inconnue sur les fonctions pas nécessairement convexes $f_s : \mathbb{R}^d \to \mathbb{R}$.

Description. En considérant un système distribué byzantin défini comme dans la Section 1.2, où chaque travailleuse a accès à un nombre fini d'échantillons de fonctions qui suivent une loi inconnue $\mathcal{D}$. Les auteurs procèdent en itérations structurées comme suit : les travailleuses calculent chacune un vecteur gradient stochastique g_i puis l'envoie de manière synchronisée à la coordinatrice. Une fois les P gradients stochastiques reçus par la coordinatrice, cette dernière met à jour le paramètre comme suit :

$$w_{k+1} = \arg \min_{y:\|y-w_1\|\leq D} \left\{ \frac{1}{2}\|y - w_k\|^2 + \eta\langle \xi_k, y - w_k \rangle \right\}$$

L'égalité ci-dessus découle directement de la méthode de descente de gradient projetée et le ξ_k sera détaillé ci-dessous.

Contribution. D. Alistarh et al.[16] fournissent des bornes supérieures et inférieures [3] qui font de leur algorithme le meilleur [4] algorithme de descente de gradient distribué résilient aux byzantines. À partir de la façon dont sont calculés les vecteurs gradients et des bornes obtenues, leur algorithme :

1. obtient une complexité d'échantillons optimale : elle est mesurée comme étant le nombre de fonctions accédées. Elle s'obtient en trouvant la complexité analytique de la méthode d'optimisation.
2. obtient un nombre optimal de calcul du gradient stochastique : ce nombre optimal est atteint car à chaque itération et pour chaque travailleuse, n'est sélectionné qu'un seul échantillon pour le calcul du gradient.
3. correspond à la complexité d'échantillons et à la complexité temporelle (qui est déterminée par le nombre d'itérations) de l'algorithme classique de DGS lorsque $\epsilon \to 0$.
4. $(1. - 3.)$ restent vraies même dans les problèmes à grande dimension. Cela se justifie par le fait que les bornes supérieures ne dépendent pas de la dimension du problème.

Pour atteindre ces quatre caractéristiques mentionnées ci-dessus, D. Alistarh et al.[16]ont émis une hypothèse forte qui est de supposer que $\|\nabla f_s(w) - \nabla f(w)\| \leq \mathcal{V}$. Contrairement à celles de P. Blanchard et al.[14] et Q. Xia et al.[2] qui supposent que $\nabla f_s(w) - \nabla f(w)$ a une variance et mesure d'asymétrie bornées.

Algorithme. ByzantineSGD forme à chaque itération un ensemble de bonnes machines noté $\mathsf{good}_k \in [P]$ puis exécute la DGP sur cet ensemble en utilisant

$$\xi_k = \frac{1}{P} \sum_{p \in |\mathsf{good}_k|} \nabla_{k,p} \qquad \text{où} \qquad \nabla_{k,p} = \nabla f_s(w_k) \quad \text{avec } k \in [T].$$

3. Uniquement les bornes supérieures nous intéresse.

4. Meilleur uniquement sous le plan théorique. L'algorithme n'a pas encore été implémenté donc ne peut être considéré plus meilleur que FABA en pratique.

Pour la formation de good_k, deux séquences d'estimations sont utilisées :

$$A_p = \sum_{t=0}^{k} \langle \nabla_{t,p}, w_t - w_0 \rangle \qquad \text{et} \qquad B_p = \sum_{t=0}^{k} \nabla_{t,p}.$$

Algorithme 6 ByzantineSGD

Entrée :

η_k : taux d'apprentissage, supérieur à 0.
T : nombre d'itérations.
w_0 : valeur initiale.
D : diamètre.
τ_A et τ_B : seuils tous supérieurs à 0.

1. $\mathsf{good}_{-1} \leftarrow [P]$;
2. **Pour** $k \leftarrow 0$ **à** $T-1$ **faire**
3. **Pour** $p \leftarrow 1$ **à** P **faire**
4. Reçoit (la coordinatrice) $\nabla_{k,p}$, $p \in [P]$;
5. $A_p = \sum_{t=0}^{k} \langle \nabla_{k,p}, w_t - w_0 \rangle$ et $B_p = \sum_{t=0}^{k} \nabla_{t,p}$;
6.
7. $A_{\mathsf{med}} \leftarrow \mathsf{median}\{A_1, A_2, \ldots, A_P\}$;
8. $B_{\mathsf{med}} \leftarrow B_p$ où $p \in [P]$ tel que $|\{j \in [P] : \|B_j - B_p\| \leq \tau_B\}| > \frac{P}{2}$;
9. $\nabla_{\mathsf{med}} \leftarrow \nabla_{k,p}$ où $p \in [P]$ tel que $|\{j \in [P] : \|\nabla_{k,j} - \nabla_{k,p}\| \leq 2\mathcal{V}\}| > \frac{P}{2}$;
10. $\mathsf{good}_k \leftarrow \left\{p \in \mathsf{good}_{k-1} : |A_p - A_{\mathsf{med}}| \leq \tau_A \wedge \|B_p - B_{\mathsf{med}}\| \leq \tau_B \wedge \|\nabla_{k,p} - \nabla_{\mathsf{med}}\| \leq 4\mathcal{V}\right\}$;
11. $w_{k+1} = \arg\min_{y:\|y-w_1\| \leq D} \left\{ \frac{1}{2}\|y - w_k\|^2 + \eta_k \langle \frac{1}{P} \sum_{p \in \mathsf{good}_k} \nabla_{k,p}, y - w_k \rangle \right\}$;

En sortie : w_T

Les auteurs ne fournissent pas d'implémentation de cet algorithme en fonction du conditionnement de la fonction objective. Les résultats déterminent juste les bornes supérieures et inférieures de la DGP.

TABLE 2.1 – Comparaison entre les complexités et vitesses de convergence des méthodes de DG centralisé et distribué.

Algorithme	$\|\nabla f\| \leq G$	**Lipschitz**	**Lipschitz et fortement convexe**	**Coût de calcul du gradient**
		Centralisé		
GD	$O(\frac{1}{\varepsilon^2})$	$O(\frac{1}{\varepsilon})$	$O\left(\log(\frac{1}{\varepsilon})\right)$	$O(nd)$
SGD	$O(\frac{1}{\varepsilon^2})$		$O\left(\frac{1}{\varepsilon}\right)$	$O(d)$
SGD mini bacth	$O(\frac{1}{\varepsilon^2})$		$O\left(\frac{1}{\varepsilon}\right)$	$O(md)$
		Distribué		
Krum	$O(dP^2)$	$O(dP^2)$	$O(dP^2)$	$O(md)$
FABA	$O(dP^2)$	$O(dP^2)$	$O(dP^2)$	$O(md)$
ByzantineSGD		$O\left(\frac{P}{\varepsilon} + \frac{1}{\varepsilon^2} + \frac{\epsilon^2 P^3}{\varepsilon^2}\right)$	$O\left(\frac{P}{\sigma} + \frac{1}{\sigma\varepsilon} + \frac{\epsilon^2 P^3}{\sigma\varepsilon}\right)$	$O(d)$

2.5 Algorithme K-means distribué

Le travail de ce mémoire porte sur l'algorithme K-means (encore appelé algorithme Lloyd) distribué qui minimise l'erreur de quantification (2.9). Cette erreur de quantification est déterminée soit lorsque la perte entre les itérations ne diminue plus ou soit lorsque le nombre d'itérations maximal est atteint.

L'erreur de quantification et la convergence de K-means peuvent être déterminées par la méthode de descente de gradient et par la méthode de Newton. En effet, L. Bottou et al.[18] commence par réécrire le problème (2.9) comme :

$$f(m) = \sum_{i=1}^{n} (x_i - m_{s_i(m)})^2$$

où $s_i(m)$ dénote le centroïde (cluster) le plus proche de x_i. Ainsi, en posant $\Delta m = -\eta_t \frac{\partial f}{\partial m}$, ils obtiennent

$$\Delta m_k = \sum_{i=1}^{n} \begin{cases} 2\eta_t(x_i - m_k) & \text{si } k = s_i(m) \\ 0 & \text{sinon.} \end{cases}$$

Cette relation nous permet d'envisager l'application des résultats précédents sur la descente de gradient à l'algorithme de K-means distribué.

Le clustering ou l'analyse de clusters est une tâche qui consiste à regrouper, à scinder une collection d'objets en un nombre fini de groupes appelés clusters ; de telle sorte que les objets qui appartiennent à un même cluster sont plus similaires que ceux appartenant aux autres clusters. Le clustering est la technique la plus utilisé en apprentissage non-supervisé car il permet de classer les objets très rapidement et sans l'aide d'un expert.

Type. Il existe plusieurs algorithmes de clustering, et ces algorithmes peuvent être classés selon leurs types à savoir :

- **Hard clustering :** ici chaque objet est attribué à un et un seul cluster. C'est le cas par exemple de K-means et de K-médoïdes.
- **Soft clustering :** ici, un objet peut appartenir à plusieurs clusters. C'est le cas de l'algorithme Expectation-maximization(EM).

Similitude. Dans la plupart des méthodes de clustering, les objets sont attribués à un cluster grâce à une mesure de similarité ou de dissimilarité. Cette mesure n'est nulle autre qu'une distance. C'est cette distance qui définit la similarité ou dissimilarité entre clusters et/ou objets.

On appelle distance sur un ensemble E non vide toute application d de E^2 dans $\mathbb{R}$ qui vérifie les conditions suivantes :

1. $\forall\ (x, y) \in E^2,\ d(x, y) \geq 0$;
2. $\forall\ (x, y) \in E^2,\ (d(x, y) = 0 \iff x = y)$;
3. $\forall\ (x, y) \in E^2,\ d(x, y) = d(y, x)$;
4. $\forall\ (x, y, z) \in E^3,\ d(x, y) \leq d(x, z) + d(z, y)$.

Si la propriété (4.) n'est pas vérifiée alors nous avons une dissimilarité, et lorsque (4.) est remplacée par $d(x, y) \leq \max\{d(x, z), d(z, y)\}$, on a affaire à un **ultra-métrique** qu'on appelle encore forte inégalité triangulaire ou métrique non-archimédienne ou

super-métrique.

Quelques mesures de similarité.

- La distance de Manhattan : $\|x-y\|_1 = \sum_i |x_i - y_i|$
- La distance Euclidienne : $\|x-y\|_2 = \sqrt{\sum_i (x_i - y_i)^2}$
- La distance Euclidienne au carré : $\|x-y\|_2^2 = \sum_i (x_i - y_i)^2$
- La distance maximum : $\|x-y\|_\infty = \max_i\{|x_i - y_i|\}$
- La similarité cosinus normalisée : $s(x,y) = cos(\theta) = \frac{\langle x,y\rangle}{\|x\|\|y\|}$
- La distance Mahalanobis : $d(x,y) = \sqrt{(x-y)^\top S^{-1}(x-y)}$, où S est la matrice de covariance.

K-means. L'idée de former K groupes d'objets à partir des centroïdes a été donné pour la première fois par H. Steinhaus[19]. Le terme K-means apparaîtra plus tard et pour la première fois dans les travaux de J. Macqueen[20] qui étudie formellement l'algorithme K-means. K-means clustering est une méthode permettant de partitionner les données en K groupes d'objets similaires à partir des centroïdes. Il s'exécute sur des objets ou données non labélisés (c'est-à-dire que nous ne connaissons pas les y_i).

L'objectif de K-means est de minimiser la variance ou l'erreur de quantification (généralement connue sous le nom de SSQ). Supposons que l'on veut former K clusters sur l'ensemble de données $\{x_i\}_{i\leq n}$, $x_i \in \mathbb{R}^d$ où le cluster $k \in [K]$ a pour centroïde $m_k \in \mathbb{R}^d$. Formellement, pour $m = (m_1, \ldots, m_K)$, minimiser SSQ équivaut à :

$$SSQ(m) = \sum_{i=1}^{n} \min_{1\leq k\leq K} (x_i - m_k)^2 \tag{2.9}$$

Pour atteindre cet objectif, après avoir fixé le nombre de centroïdes (clusters), à savoir K, l'algorithme procède de manière itérative en déplaçant les centroïdes pour minimiser la variance totale au sein des clusters. Étant donnée un ensemble d'échantillons $\{x_i\}_{i=1}^n$, K-means procède en trois étapes pour le problème (2.9) :

1. Initialisation des centroïdes (clusters) ;
2. Pour chaque centroïde (cluster), on identifie le sous-ensemble des points qui est plus proche du centroïde en question que les autres centroïdes (clusters) ;
3. Calculer les vecteurs moyennes de ces sous-ensembles de points puis attribuer aux clusters les nouveaux centroïdes (les vecteurs moyennes précédemment calculés) ;

Les étapes (2.) et (3.) sont soit itérées jusqu'à convergence (c'est-à-dire que les centroïdes trouvés à une itération t sont quasiment les mêmes que ceux trouvés à l'itération $t+1$), soit arrêtées après un nombre fixe d'itérations. Généralement, les K centroïdes initiaux sont choisis dans les données de manière aléatoire.

L'algorithme de K-means[21] est le suivant :

Algorithme 7 K-means

Entrée :
$\{x_i\}_{i=1}^n$: jeu de données, $x_i \in \mathbb{R}^d$;
K : nombre de clusters, supérieur à 0.
T : nombre d'itérations, supérieur à 0.

1. Initialiser les centroïdes des clusters $\{m_k\}_{k=1}^K$;
2.
3. **Pour** $t \leftarrow 1$ **à** T **faire**
4. $inertie \leftarrow 0$;
5. **Pour** $k \leftarrow 1$ **à** K **faire**
6. $m'_k \leftarrow 0$; $n'_k \leftarrow 0$;
7.
8. **Pour** chaque x_i dans $\{x_i\}_{i=1}^n$ **faire**
9. **Pour** $k \leftarrow 1$ **à** K **faire**
10. Calculer une mesure de similarité entre x_i et m_k ;
11.
12. Trouver le centroïde m_l le plus proche de x_i ;
13. $m'_l \leftarrow m'_l + x_i$; $n'_l \leftarrow n'_l + 1$;
14. $inertie \leftarrow inertie + d^2(x_i, m_l)$;
15.
16. **Pour** $k \leftarrow 1$ **à** K **faire**
17. $n'_k \leftarrow \max(n'_k, 1)$; $m_k \leftarrow m'_k / n'_k$;

En sortie : $\{m_k\}_{k=1}^K$

En pratique, il est courant de commencer avec de nombreuses initialisations différentes, d'exécuter K-means pour chacune, puis de sélectionner l'initialisation qui a la plus petite erreur.

L'Algorithme 7 a une complexité de calcul de $O(nKdT)$. Ce qui fait de lui un algorithme peu coûteux (si n et d ne sont pas très grands), rapide en terme de convergence et adapté aux données creuses[21]. Néanmoins, la convergence de l'Algorithme 7 dépend beaucoup de la phase d'initialisation qui est, en elle-même un algorithme. Dans la littérature scientifique, on peut citer trois algorithmes d'initialisation tels que :

- *random*. C'est la méthode la plus naïve de toutes les méthodes d'initialisation. Elle consiste à choisir les centroïdes de manière aléatoire dans $\{x_i\}_{i=1}^n$.
- *iter*[22]. Dans cette méthode l'initialisation se déroule comme suit : (1) Le premier centroïde est choisi comme le point qui est le plus éloignée du vecteur zéro. Même calcul pour le deuxième centroïde mais cette fois ci avec le premier centroïde. (2) Le troisième centroïde est choisi comme le point qui a la somme maximale des carrés des distances entre lui et le premier centroïde, et lui et le deuxième centroïde. Répète ce processus avec les deux derniers centroïdes jusqu'à obtenir le nombre de centroïdes voulu.
- *kmeans++*[23]. Après avoir sélectionné uniformément comme premier centroïde un point dans $\{x_i\}_{i=1}^n$, cette méthode sélectionne les centroïdes comme étant les points ayant la distance la plus courte avec les centroïdes les plus proches déjà sélectionnés.

K-means distribué. Malgré son faible coût computationnel et sa vitesse de convergence, K-means devient très coûteux lorsque n et d sont grands. Pour palier cela, une solution est de répartir la charge de travail. Dhillon et al.[21] propose une implémentation parallèle de K-means basée sur le modèle SPMD en utilisant la

progammation par passage de message à l'aide de la bibliothèque MPI.

Description. Soit un système distribué défini comme dans la Section 1.2 mais sans machines byzantines, c'est-à-dire $\epsilon = 0$. Avant le début de l'exécution de K-means distribué, la coordinatrice initialise les centroïdes puis répartit de manière aléatoire les données $\{x_i\}_{i=1}^n$, $x_i \in \mathbb{R}^d$ aux travailleuses de telle sorte que chaque travailleuse $p \in [P]$ ait à sa disposition un jeu de données X_p, avec $\{x_i\}_{i=1}^n = X_1 + X_2 + \cdots + X_P$. À chaque itération $t \in [T]$, sur le jeu de données qui lui a été assigné, chaque travailleuse p calcule K vecteurs réels $\{m_k'^{(p)}\}_{k=1}^K$ où, $m_k'^{(p)}$ est la somme des points qui se trouvent dans le cluster k, et K scalaires entiers $\{n_k'^{(p)}\}_{k=1}^K$ où, $n_k'^{(p)}$ est le nombre de points contenu dans le cluster k. Après que les KP vecteurs sommes et scalaires aient été calculés, les p travailleuses envoient leurs quantités à la coordinatrice. Cette dernière se charge ensuite de les agréger et, les K centroïdes résultants de cette agrégation sont considérés comme les nouveaux centroïdes pour les itérations futures.

Agrégation. Lorsque la coordinatrice reçoit les KP vecteurs sommes et scalaires, elle les agrège en faisant une moyenne arithmétique : en sommant les vecteurs sommes qui appartiennent au même cluster, en sommant les scalaires qui appartiennent au même cluster puis en divisant les vecteurs sommes résultants par les scalaires résultants associés au même cluster. On obtient les K centroïdes suivants :

$$\left\{ \frac{m_1'^1 + m_1'^2 + \cdots + m_1'^P}{n_1'^1 + n_1'^2 + \cdots + n_1'^P}, \cdots, \frac{m_K'^1 + m_K'^2 + \cdots + m_K'^P}{n_K'^1 + n_K'^2 + \cdots + n_K'^P} \right\}$$

Algorithme 8 K-means distribué

Entrée :
$\{x_i\}_{i=1}^n$: jeu de données, $x_i \in \mathbb{R}^d$;
K : nombre de clusters, supérieur à 0.
T : nombre d'itérations, supérieur à 0.

1. **Si** $ID = 0$ **alors**
2. $inertie \leftarrow 0$;
3. Initialiser les centroïdes des clusters $\{m_k\}_{k=1}^K$;
4. Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;
5. Répartir $\{x_i\}_{i=1}^n$ aux travailleuses ;
6. **Sinon**
7. $som\_inertie \leftarrow 0$;
8.
9. **Pour** $t \leftarrow 1$ **à** T **faire**
10.
11. **Pour** $k \leftarrow 1$ **à** K **faire**
12. $m_k' \leftarrow 0_{\mathbb{R}^d}$; $n_k' \leftarrow 0$;
13.
14. **Si** $ID \neq 0$ **alors**
15. **Pour** chaque x_i dans X_p **faire**
16. **Pour** $k \leftarrow 1$ **à** K **faire**
17. Calculer une mesure de similarité entre x_i et m_k ;
18.

19. Trouver le centroïde m_l le plus proche de x_i ;
20. $m'_l \leftarrow m'_l + x_i$; $n'_l \leftarrow n'_l + 1$;
21. $som\_inertie \leftarrow som\_inertie + d^2(x_i, m_l)$;
22.
23. **Pour** $k \leftarrow 1$ **à** K **faire**
24. Diffuser m'_k, n'_k et $som\_inertie$ à la coordinatrice ;
25.
26. **Si** $ID = 0$ **alors**
27. **Pour** $k \leftarrow 1$ **à** K **faire**
28. $n'_k \leftarrow$ somme des n'_k envoyés par les travailleuses ;
29. $m'_k \leftarrow$ somme des m'_k envoyés par les travailleuses ;
30. $n'_k \leftarrow \max(n'_k, 1)$;
31. $m_k \leftarrow m'_k / n'_k$;
32.
33. $inertie \leftarrow$ somme des $som\_inertie$ envoyés par les travailleuses ;
34. Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;

En sortie : $\{m_k\}_{k=1}^K$

L'algorithme K-means distribué[21] a une complexité de calcul égale à $O(\frac{ndKT}{P})$ et une complexité de communication égale à $O(dKT)$.

Des travaux récents ont été mené pour développer des algorithmes de K-means clustering distribué. Comme exemple, on peut citer les travaux de G. Olivia[24] dans lesquels est proposé un algorithme de K-means distribué à base de consensus où les nœuds ne peuvent communiquer qu'avec leurs voisins directes. Nous utilisons K-means distribué[21] car il est facile à comprendre, à implémenter et repose sur une topologie très utilisée.

TABLE 2.2 – Comparaison entre les complexités de K-means et celles de K-means distribué.

Algorithme	**Coût de calcul**	**Coût de communication**
K-means	$O(ndKT)$	0
K-means disribué	$O\left(\frac{ndKT}{P}\right)$	$O(dKT)$

CHAPITRE 3

K-MEANS DISTRIBUÉ RÉSILIENT

Dans les précédents chapitres, nous nous sommes attardés sur la description d'un système distribué (en AA et dans un environnement distribué) tout en présentant au passage l'algorithme de DG et quelques algorithmes de DG distribué ainsi que K-means et K-means distribué. Dans ce chapitre, nous traitons des erreurs engendrées par K-means distribué byzantin et de la méthode apportée pour corriger ces erreurs afin qu'il puisse toujours converger même en présence des byzantines. Cette correction se fera à l'aide de la méthode FABA[2].

3.1 K-means byzantin

Dans cette section, K-means byzantin désigne un algorithme K-means qui s'exécute dans un environnement distribué possédant des machines byzantines.

Description. Considérons un système distribué défini comme dans la Section 1.2. Après que la coordinatrice ait initialisé les centroïdes puis réparti de manière aléatoire les données à chaque travailleuse $p \in [P]$, de telle sorte que chaque travailleuse ait à sa disposition un jeu de données X_p avec $\{x_i\}_{i=1}^n = X_1 + X_2 + \cdots + X_P$, tout comme K-means distribué, K-means byzantin procède comme suit : à chaque itération $t \in [T]$, chaque bonne travailleuse $i \in \mathcal{G}$ calcule K vecteurs réels $\{m_k'^{(i)}\}_{k=1}^K$ et K scalaires entiers $\{n_k'^{(i)}\}_{k=1}^K$ et chaque byzantine $b \in \mathcal{B}$ génère/produit/calcule K vecteurs réels potentiellement défavorables $\{m_k'^{(b)}\}_{k=1}^K$ et K scalaires entiers $\{n_k'^{(b)}\}_{k=1}^K$. Les KP vecteurs et scalaires calculés, les p travailleuses envoient de manière synchronisée leurs quantités à la coordinatrice qui se charge par la suite de les agréger. Les K centroïdes résultants de cette agrégation sont considérés comme les nouveaux centroïdes pour les itérations futures.

Génération des byzantins. Le but ici est de fournir des procédés qui vont nous permettre de générer des nœuds byzantins. Ces procédés devront générer des valeurs (centroïdes) difficiles à détecter par la règle d'agrégation mise en place. Pour cela, nous proposons deux méthodes de génération de byzantins : *méthode 1* et *méthode 2*.

Méthode 1. Pour générer les centroïdes byzantins, tous les $b \in \mathcal{B}$ calculerons d'abord leurs quantités $\{m_k'^{(b)}\}_{k=1}^K$ et $\{n_k'^{(b)}\}_{k=1}^K$ comme s'ils appartenaient à $\mathcal{G}$ puis procédera comme suit : $\forall\ b \in \mathcal{B}$, b modifie ses K vecteurs sommes $\{m_k'^{(b)}\}_{k=1}^K$ en choisissant (de manière aléatoire) comme vecteur somme de chaque cluster un point appartenant à ce cluster.

Méthode 2. Dans cette méthode, nous générons les machines byzantines comme suit : $\forall\ b \in \mathcal{B}$, b modifie ses centroïdes en éliminant $\frac{1}{3}$ des points dans chaque cluster de

manière uniforme puis recalcule les centroïdes sur les $\frac{2}{3}$ restants.

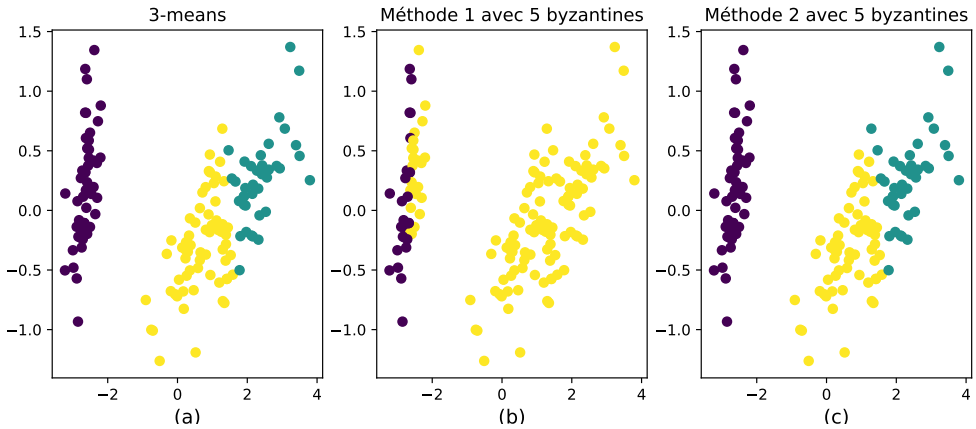


FIGURE 3.1 – Sur un environnement distribué de 10 travailleuses et le jeu de données Iris, (a) représente la formation de 3 clusters sans byzantines. (b) représente la formation de 3 clusters avec 5 byzantines générées par *méthode 1* (la *méthode 1* a biaisé les centroïdes finaux car nous en avons 2) et (c) la formation de 3 clusters avec 5 byzantines générées par *méthode 2* (la *méthode 2* n'a pas biaisé les centroïdes finaux, nous en avons toujours 3 qui sont semblables à ceux de (a)).

Nous avons choisi d'utiliser *méthode 1* car non seulement elle perturbe bien les centroïdes, mais aussi, elle évite d'avoir des centroïdes générés qui sont trop éloignés des vrais centroïdes (contrairement à *méthode 2* qui génère des centroïdes proches des vrais) et facilement détectables par les règles d'agrégation. La Figure 3.2 illustre cela.

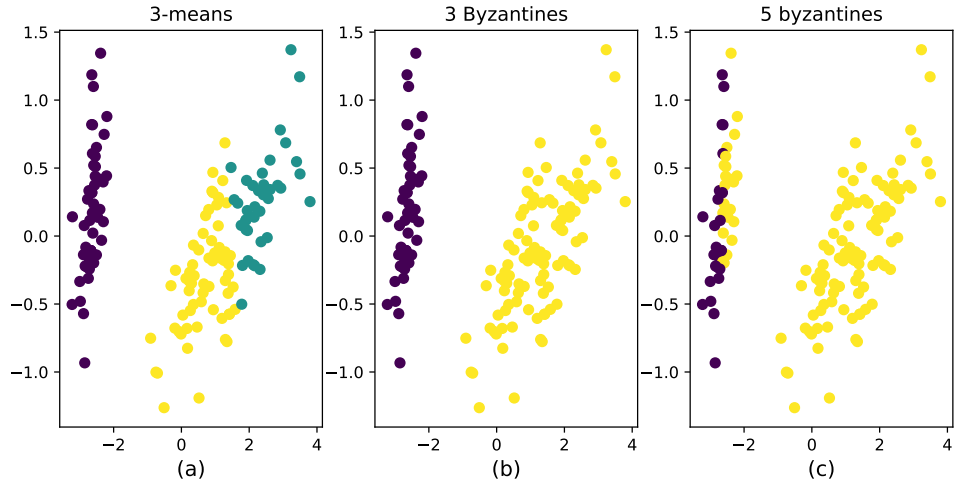


FIGURE 3.2 – Sur un environnement distribué de 10 travailleuses et le jeu de données Iris, (a) représente la formation de 3 clusters sans byzantines. (b) 3 clusters avec 3 byzantines et (c) 3 clusters avec 5 byzantines. Génération faite par la *methode 1*.

Algorithme 9 K-means byzantin

Entrée :
$\{x_i\}_{i=1}^n$: jeu de données, $x_i \in \mathbb{R}^d$;
K : nombre de clusters, supérieur à 0.
T : nombre d'itérations, supérieur à 0.

1. **Si** $ID = 0$ **alors**
2. $inertie \leftarrow 0$;
3. Initialiser les centroïdes des clusters $\{m_k\}_{k=1}^K$;
4. Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;
5. Répartir $\{x_i\}_{i=1}^n$ aux travailleuses ;
6. **Sinon**
7. $som\_inertie \leftarrow 0$;
8.
9. **Pour** $t \leftarrow 1$ **à** T **faire**
10.
11. **Pour** $k \leftarrow 1$ **à** K **faire**
12. $m'_k \leftarrow 0_{\mathbb{R}^d}$; $n'_k \leftarrow 0$;
13.
14. **Si** $ID \neq 0$ **alors**
15. **Pour** chaque x_i dans X_p **faire**
16. **Pour** $k \leftarrow 1$ **à** K **faire**
17. Calculer une mesure de similarité entre x_i et m_k ;
18.
19. Trouver le centroïde m_l le plus proche de x_i ;
20. $m'_l \leftarrow m'_l + x_i$; $n'_l \leftarrow n'_l + 1$;
21. $som\_inertie \leftarrow som\_inertie + d^2(x_i, m_l)$;
22.
23. **Si** $ID \in \mathcal{B}$ **alors**
24. **Pour** $k \leftarrow 1$ **à** K **faire**
25. $m'_k \leftarrow$ Un x_i contenue dans le cluster k ;
26.
27. **Si** $ID \neq 0$ **alors**
28. **Pour** $k \leftarrow 1$ **à** K **faire**
29. Diffuser m'_k, n'_k et $som\_inertie$ à la coordinatrice ;
30.
31. **Si** $ID = 0$ **alors**
32. **Pour** $k \leftarrow 1$ **à** K **faire**
33. $n'_k \leftarrow$ somme des n'_k envoyés par les travailleuses ;
34. $m'_k \leftarrow$ somme des m'_k envoyés par les travailleuses ;
35. $n'_k \leftarrow \max(n'_k, 1)$;
36. $m_k \leftarrow m'_k / n'_k$;
37.
38. $inertie \leftarrow$ somme des $som\_inertie$ envoyés par les travailleuses ;
39. Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;

En sortie : $\{m_k\}_{k=1}^K$

L'Algorithme 9 K-means byzantin diffère de l'Algorithme 8 qu'au niveau de la génération des erreurs byzantines. Le changement se trouve de la ligne 26 à 28 ; ces lignes consistent uniquement à générer les quantités suivant le procédé énoncé précédemment.

3.2 K-means résilient aux fautes byzantines

Sachant la relation qu'il y a entre K-means et les méthodes de DG (Section 2.5). Pour corriger ces erreurs byzantines, nous proposons d'utiliser la règle d'agrégation FABA à K-means byzantins. De ce fait, supposons que

Hypothèse 3.1. *Chaque centroïde k est associé ou peut être vu comme un vecteur gradient.*

Description. Soit un système distribué défini comme dans la Section 1.2. Après répartition des données aux travailleuses et initialisation des centroïdes, K-means résilient aux byzantins procède comme suit : à chaque itération $t \in [T]$, chaque bonne travailleuse $i \in \mathcal{G}$ calcule K centroïdes $\{m_k'^{(i)}\}_{k=1}^K$ et chaque byzantine $b \in \mathcal{B}$ génère/produit/calcul K centroïdes potentiellement malveillantes $\{m_k'^{(b)}\}_{k=1}^K$. Les KP centroïdes sont ensuite envoyés à la coordinatrice qui les agrégera. les K centroïdes résultants de cette agrégation sont considérés comme les nouveaux centroïdes pour les itérations futures. La méthode de génération des centroïdes byzantins est la *methode 1* décrite dans la Section 3.1.

Agrégation. Lorsque la coordinatrice reçoit les KP centroïdes à chaque itération $t \in [T]$, elle les agrège en appliquant FABA sur les centroïdes de même cluster. On obtient alors les K centroïdes suivants :

$$\left\{\mathsf{FABA}(m_1'^1 + m_1'^2 + \cdots + m_1'^P), \cdots, \mathsf{FABA}(m_K'^1 + m_K'^2 + \cdots + m_K'^P)\right\}$$

Algorithme 10 K-means résilient aux byzantines

Entrée :
$\{x_i\}_{i=1}^n$: jeu de données, $x_i \in \mathbb{R}^d$;
K : nombre de clusters, supérieur à 0.
T : nombre d'itérations, supérieur à 0.

Si $ID = 0$ **alors**
  $inertie \leftarrow 0$;
  Initialiser les centroïdes des clusters $\{m_k\}_{k=1}^K$;
  Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;
  Répartir $\{x_i\}_{i=1}^n$ aux travailleuses ;
Sinon
  $som\_inertie \leftarrow 0$;

Pour $t \leftarrow 1$ **à** T **faire**

  Pour $k \leftarrow 1$ **à** K **faire**
    $m_k' \leftarrow 0_{\mathbb{R}^d}$; $n_k' \leftarrow 0$;

  Si $ID \neq 0$ **alors**
    Pour chaque x_i dans X_p **faire**
      Pour $k \leftarrow 1$ **à** K **faire**
        Calculer une mesure de similarité entre x_i et m_k ;

      Trouver le centroïde m_l le plus proche de x_i ;
      $m_l' \leftarrow m_l' + x_i$; $n_l' \leftarrow n_l' + 1$;
      $som\_inertie \leftarrow som\_inertie + d^2(x_i, m_l)$;

23. **Si** $ID \in \mathcal{G}$ **alors**
24. **Pour** $k \leftarrow 1$ **à** K **faire**
25. $m'_k \leftarrow m'_k / \max(n'_k, 1)$;
26. Diffuser $\{m'_k\}_{k=1}^K$ et *som_inertie* à la coordinatrice ;
27.
28. **Si** $ID \in \mathcal{B}$ **alors**
29. **Pour** $k \leftarrow 1$ **à** K **faire**
30. $m'_k \leftarrow$ Un x_i contenue dans le cluster k ;
31. Diffuser $\{m'_k\}_{k=1}^K$ et *som_inertie* à la coordinatrice ;
32.
33. **Si** $ID = 0$ **alors**
34. **Pour** $k \leftarrow 1$ **à** K **faire**
35. $m_k \leftarrow \mathsf{FABA}(\{m'^1_k, m'^2_k, \ldots, m'^P_k\}, \epsilon P)$;
36.
37. *inertie* $\leftarrow$ somme des *som_inertie* envoyés par les travailleuses ;
38. Diffuser $\{m_k\}_{k=1}^K$ aux travailleuses ;

En sortie : $\{m_k\}_{k=1}^K$

Une fois FABA couplée avec K-means distribué, l'on peut maintenant passer à la partie implémentation et résultats pour voir comment se comporte K-means.

3.3 Expérimentations

Nous avons implémenté et exécuté K-means byzantin couplé avec FABA en simulant un environnement distribué à l'aide de MPI.

3.3.1 Environnement matériel et logiciel

Côté matériel, l'exécution de K-means byzantin + FABA s'est faite sur un modeste core i3 Dell Optiplex de 04 cœurs cadencés à 3.2 GHz possédant 8 Go de RAM. Côté logiciel, nous avons utilisé le système d'exploitation Ubuntu 16.04 et MPI pour simuler un environnement distribué. MPI est un protocole de communication conçu pour les calculs dans des architectures parallèles ou distribuées, où la communication peut se faire soit par point-à-point soit par diffusion totale.

Dans nos expériences, nous simulons un système distribué contenant 11 machines dont une coordinatrice et 10 travailleuses dans lequel nous effectuons des corrections où on a 20% puis 40% de machines byzantines.

3.3.2 Jeu de données et résultats

Pour mener les expérimentations, l'on a aussi besoin des données, données sur lesquelles s'exécutera K-means. Pour se faire, nous avons choisi 03 jeux de données :

Iris [1] **:** est un jeu de données très populaire dans la reconnaissance des formes. Il contient 04 caractéristiques, 03 classes de 50 échantillons chacun où chaque classe dénote un type de plante Iris. Il sera question pour nous de former 03 groupes de données ; 03 parce que nous savons qu'il existe uniquement 03 classes dans les données, bien que K-means est un algorithme de type apprentissage non-supervisé.

1. https ://archive.ics.uci.edu/ml/datasets/iris

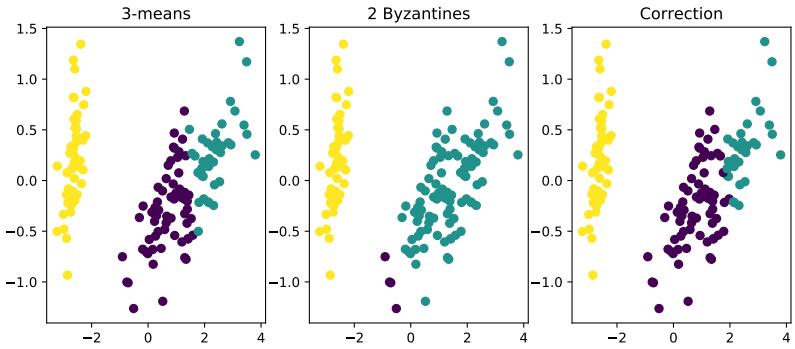


FIGURE 3.3 – 3-means, 2-byzantins, avec correction.

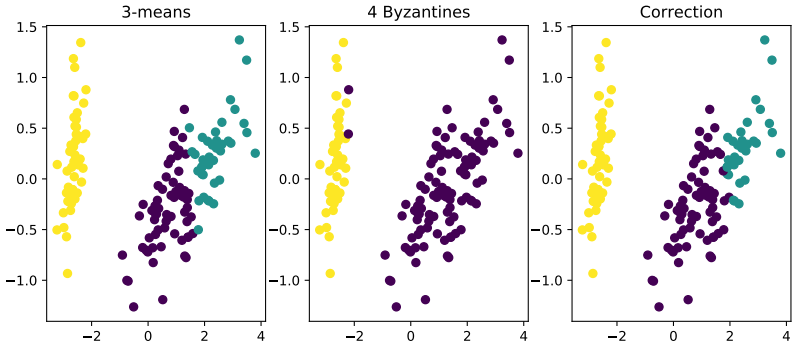


FIGURE 3.4 – 3-means, 4-byzantins, avec correction.

K-means byzantin + FABA offrent de très bons résultats sur Iris. Même quand les byzantines arrivent à complètement perturber les clusters, la correction est un succès. Même quand nous avons 9 machines byzantines ($\epsilon = 0.9$) qui pertubent les centroïdes, FABA arrive toujours à corriger les erreurs. Pourtant FABA est censé fonctionner lorsque $\epsilon < 0.5$.

Vehicle Silhouettes [2] **:** est un jeu de données dont la tâche principale est de classifier certains véhicules. Ce jeu de données contient 21 caractéristiques dont 20 variables explicatives et 1 variable réponse. Les 20 variables explicatives dénotent les caractéristiques qui ont été extraites des silhouettes des véhicules ainsi que des caractéristiques utilisant à la fois des mesures classiques basées sur les moments telles que la variance à l'échelle, l'asymétrie et l'aplatissement autour des axes et des mesures tels que les creux, la circularité et la compacité. La variable résponse a 4 classes dont OPEL, SAAB, BUS et VAN. En ne considerant pas la variable réponse, il sera question pour nous de former 4 clusters dans le but de retrouver les vraies classes précédement supprimées.

2. https ://archive.ics.uci.edu/ml/datasets/Statlog+%28Vehicle+Silhouettes%29

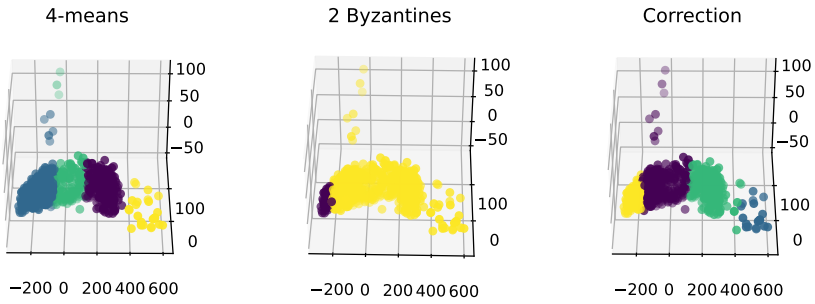


FIGURE 3.5 – 4-means, 2-byzantins, avec correction.

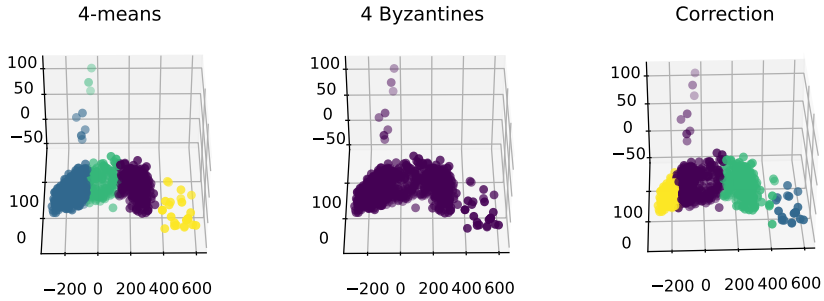


FIGURE 3.6 – 4-means, 4- byzantins, avec correction.

Ici aussi, on arrive à reconstituer les clusters presque parfaitement. Même lorsque $\epsilon = 0.9$.

Breast Cancer Wisconsin [3] **:** est un jeu de données pour classification. Ce jeu de données classe les personnes comme des personnes ayant ou non une tumeur bénigne. Il contient 11 caractéristiques dont 10 variables explicatives et 1 variable réponse. À partir des variables explicatives, il sera question pour nous de former 2 clusters qui réprésentera des personnes ayant et n'ayant pas de tumeur.

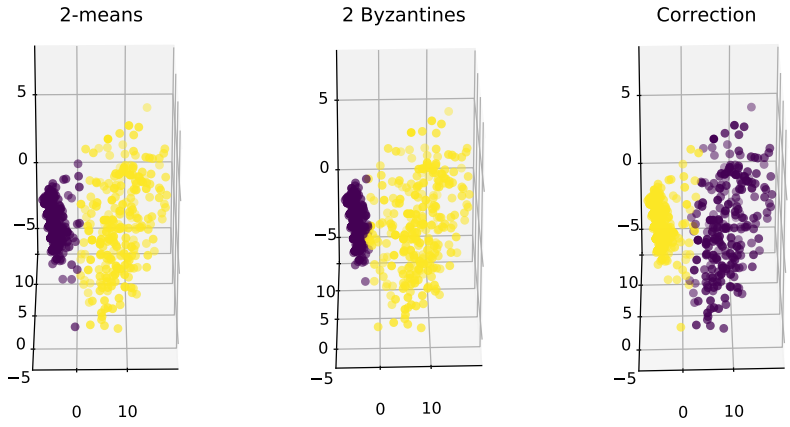


FIGURE 3.7 – 4-means, 2-byzantins, avec correction.

3. https ://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)

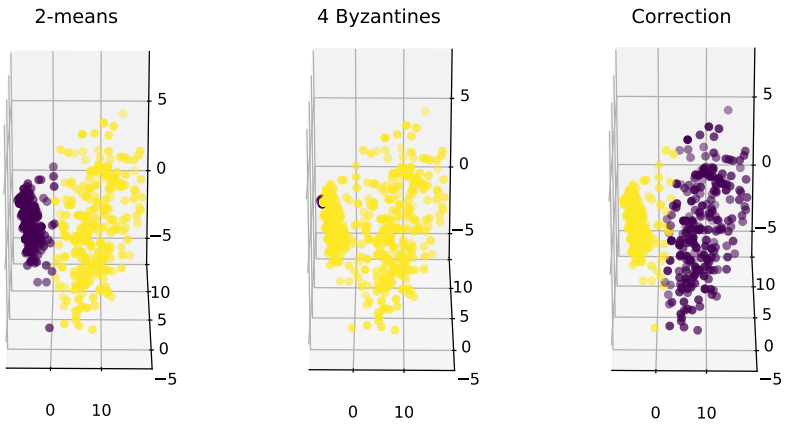


FIGURE 3.8 – 4-means, 4- byzantins, avec correction.

Les mêmes phénomènes observés dans les 2 précédentes expériences surviennent aussi ici.

Nous avons aussi exécuté l'Algorithme 10 sur des données de grande taille : enron [4], abcnews [5] et covtype [6]. Nous y avons rencontré ici des problèmes au niveau du temps de calcul. En effet, lorsqu'exécuté sur des données volumineuses, l'Algorithme 10 prend un temps prohibitif pour pouvoir fournir les résultats et faute de machines puissantes nous nous sommes retournés vers des données de petites tailles.

3.4 Discussions

À travers les travaux menés lors de cette étude, sans apporter de preuve formelle sur la convergence de K-means couplé avec FABA, dans un environnement byzantin, le couple K-means et FABA permet de corriger les erreurs byzantines et offre un algorithme de clustering efficace et stable. Efficace parce qu'il renvoie des clusters assez proches des vrais clusters [7] et cela même lorsque $\epsilon > 1/2$. Stable parce que les clusters renvoyés sont quasiment les mêmes peu importe la portion de machines byzantines choisie.

Toutefois, des 3 expériences menées ci-dessus, on observe un défaut à savoir : la correction des centroïdes lorsque ϵ tend vers 1. En effet, il n'est pas normal que lorsque $\epsilon = 0.9$ l'on puisse toujours bien corriger. Ce phénomène peut s'expliquer par le fait que, dû à la génération des byzantines (qui prend comme centroïde byzantin un point appartenant au cluster), les centroïdes byzantins sont suffisamment proches des vrais centroïdes pour pouvoir permettre à l'algorithme de converger lorsqu'une partie des byzantins est supprimée. Des recherches supplémentaires sont donc nécessaires pour établir une nouvelle méthode de génération de machines byzantines pour éviter la convergence lorsque ϵ tend $\frac{1}{2}$.

On note aussi le manque de généralisation des résultats car notre étude n'est qu'expérimentale. Les futurs travaux devraient mener une étude formelle sur la convergence de K-means lorsque la règle FABA est utilisée et aussi étudier K-means couplé à l'amélioration de FABA proposé par Noutcha [15].

4. https ://www.kaggle.com/ankur561999/data-cleaning-enron-email-dataset
5. https ://www.kaggle.com/thebrownviking20/k-means-clustering-of-1-million-headlines/
6. https ://archive.ics.uci.edu/ml/machine-learning-databases/covtype/
7. "Vrais clusters" fait référence aux clusters obtenus si on avait pas de machines byzantines.

CHAPITRE 4

CONCLUSION GÉNÉRALE

Dû à la croissance très rapide des données et du manque de productivité, les algorithmes classiques d'AA, d'analyse et fouille de données ne sont plus pratiques et efficaces. De ce fait, ces dernières années, les chercheurs se sont tournés vers l'AA distribué qui consiste à concevoir et analyser les algorithmes d'AA distribué à partir des algorithmes centralisés afin de pouvoir gérer de grande quantité de données et obtenir des résultats plus rapidement. Le clustering étant l'une des principales techniques en AA et fouille de données, il a K-means comme algorithme le plus populaire et utilisé par les praticiens lors de la résolution des tâches non-supervisées. Par le billet de ce mémoire, il était question pour nous d'étudier la résilience de K-means aux erreurs byzantines dans un environnement distribué (client-serveur) possédant P machines où, une portion $\epsilon < 1/2$ d'entre elles est byzantine.

Pour aborder ce problème, il a fallu dans un premier temps définir quelques notions principales telles qu'un système distribué, une erreur byzantine et la résilience ; afin de fournir une définition formelle de ce que l'on entend par "environnement distribué byzantin". Puis, du fait que la méthode de descente de gradient soit aussi bien étudiée et analysée dans le cadre centralisé que distribué, nous avons présenter quelques algorithmes de descente de gradient centralisés (descente de gradient, descente de gradient stochastique et descente de gradient mini-batch) ainsi que quelques-uns distribués (FABA, Krum et ByzantineSGD), résilients aux machines ou erreurs byzantines. Nous y avons vu que construire un algorithme de descente de gradient distribué résilient aux byzantines dans une configuration client/serveur consiste essentiellement à concevoir une règle d'agrégation robuste (FABA, Krum et ByzantineSGD) au niveau du serveur. De plus, nous avons aussi présenté K-means centralisé et sa version distribué, tout en le reformulant en un problème de descente de gradient pour montrer ainsi le lien qu'il y a entre l'algorithme K-means clustering et l'algorithme de descente de gradient.

Après avoir montré l'impact qu'a les erreurs byzantines sur la formation des clusters, pour résoudre le problème de rendre K-means résilient aux byzantines, nous avons utilisé l'algorithme FABA (qui est un algorithme d'agrégation de vecteurs gradients calculés par les machines) pour agréger les centroïdes calculés par les différentes machines. L'utilisation de FABA comme règle d'agrégation de centroïdes dans K-means byzantin corrige les erreurs byzantines en éliminant les centroïdes byzantins, ce qui nous permet d'obtenir un algorithme de K-means distribué résilient aux byzantins. Ceci est montré à travers une étude expérimentale sur un core i3 Dell Optiplex de 04 cœurs cadencés à 3.2 GHz possédant 8 Go de RAM en utilisant les jeux de données Iris, Vehicle Silhouettes et Breast Cancer Wisconsin, sur lesquels nous avons exécuté K-means byzantin + FABA. Nous simulions un système distribué contenant 11 machines dont une coordinatrice et 10 travailleuses dans lequel nous effectuons des corrections où nous avons 20% ($\epsilon = 0.2$) puis 40% ($\epsilon = 0.4$) des machines byzantines.

♣ Bibliographie ♣

[1] J. Verbraeken, M. Wolting, J. Katzy, J. Kloppenburg, T. Verbelen, and J. S. Rellermeyer, "A survey on distributed machine learning," *ACM Comput. Surv.*, vol. 53, no. 2, 2020.

[2] Q. Xia, Z. Tao, Z. Hao, and Q. Li, "FABA :an algorithm for fast aggregation against byzantine attacks in distributed neural networks," in *International Joint Conference on Artificial Intelligence*, 2019, pp. 4824–4830.

[3] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, 9th Edition*. Wiley, 2014, ch. 17.

[4] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM (Association of Computing Machinery)*, vol. 33 :8, 8 1990.

[5] S. Saravi, R. Kalawsky, D. Joannou, M. R. Casado, G. Fu, and F. Meng, "Use of artificial intelligence to improve resilience and preparedness against adverse flood events," *Water*, vol. 11, no. 5, 2019.

[6] M. A. Heroux, "Toward resilient algorithms and applications," *arXiv preprint 1402.3809*, 2014.

[7] A. Kumar and S. Mehta, "A survey on resilient machine learning," *arXiv preprint arXiv :1707.03184*, 2017.

[8] J. Steinhardt, "Robust learning : Information theory and algorithms," Ph.D. dissertation, Stanford University, CA, USA, 2018.

[9] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4(3) :382–401, 1982.

[10] L. Bottou, F. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *arXiv preprint arXiv :1606.04838*, 2018.

[11] Y. Nesterov, *Introductory Lectures on Convex Optimization - A Basic Course*, ser. Applied Optimization. Springer, 2004, vol. 87.

[12] A. d'Aspremont, D. Scieur, and A. Taylor, "Acceleration methods," *arXiv preprint arXiv :2101.09545*, 2021.

[13] A. S. Nemirovski, A. B. Juditsky, G. Lan, and A. Shapiro, "Robust Stochastic Approximation Approach to Stochastic Programming," *SIAM Journal on Optimization*, vol. 19, no. 4, pp. 1574–1609, Jan 2009.

[14] P. Blanchard, E. E. Mahdi, R. Guerraoui, and S. Julien, "Machine learning with adversaries : Byzantine tolerant gradient descent," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[15] J. Noutcha, "Résilience des algorithmes de descente de gradient distribué," Master's thesis, Université de Yaoundé I, Yaoundé, Cameroun, 2021.

[16] D. Alistarh, Z. Allen-Zhu, and J. Li, "Byzantine stochastic gradient descent," in *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[17] S. Bubeck, "Convex optimization : Algorithms and complexity," *arXiv preprint arXiv :1405.4980*, 2015.

[18] L. Bottou and Y. Bengio, "Convergence properties of the k-means algorithms," in *Advances in neural information processing systems*, 1995, pp. 585–592.

[19] H. Steinhaus, "Sur la division des corps matériels en parties," *Bulletin de l'Académie Polonaise des Sciences, Classe 3*, vol. 4, pp. 801–804, 1957.

[20] J. Macqueen, "Some methods for classification and analysis of multivariate observations," in *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*, 1967, pp. 281–297.

[21] S. Dhillon and S. Modha, "A data-clustering algorithm on distributed memory multiprocessors," 1999.

[22] M. N.Joshi, "Parallel k-means algorithm on distributed memory multiprocessors," spring 2003.

[23] D. Arthur and S. Vassilvitskii, "K-means++ : The advantages of careful seeding." Society for Industrial and Applied Mathematics, 2007, p. 1027–1035.

[24] G. Oliva, R. Setola, and C. N. Hadjicostis, "Distributed k -means algorithm," *arXiv preprint 1312.4176v3*, 2014.

♣ Appendix ♣

A Codes

Voici l'extrait du code source python de la fonction qui nous permet de simuler les erreurs (centroïdes) byzantines.

```
    def methode_1(data, idx_by_cluster):
        """
            A method that generates byzantine errors

            Parameters :
            -----------
            data : ndarray of shape (n_samples, n_features)
                Samples to be clustered

            idx_by_cluster : ndarray of shape (n_clusters, length(X_worker))
                X_worker is the samples assigned to associated cluster
                Row 0 contains the indexes of the samples which in cluster 0
                Row 1 contains the indexes of the samples which in cluster 1
                ...

            Return :
            -------
            centroids : ndarray of shape (n_clusters, n_features)
                Byzantine centroids.
        """

        n_cluster, n_features = idx_by_cluster.shape[0], data.shape[1];
        centroids = np.zeros((n_cluster, n_features));

        for cluster in range(0, n_cluster):
            idx = idx_by_cluster[cluster];
            temp_idx = [x for x in idx if x != -1];

            if temp_idx == []:
                pass;
            else:
                idx_selected = np.random.random_integers(0, len(temp_idx)-1);
                centroids[cluster] = sparse_or_not(data[temp_idx[idx_selected]]);

        return centroids;
```

Et voici l'extrait du code source python de FABA qui nous permet de corriger les erreurs (centroïdes) byzantines.

```
    def faba(good_centroids, bad_centroids, n_byzan):
        """
            Fix byzantine (bad) centroids according FABA's method

```

```
            Parameters :
            -----------
            good_centroids : ndarray of shape (n_workers, n_clusters, n_features)
                All centroids computed by the good workers

            bad_centroids : ndarray of shape (n_workers, n_clusters, n_features)
                All centroids computed by the bad workers

            n_byzan : int
                Number of byzantine workers

            Return :
            -------
            final_centroids : ndarray of shape (n_clusters, n_features)
                Merged centroids
        """

        all_centroids = np.vstack((good_centroids, bad_centroids));
        agregated_centroids = np.zeros((good_centroids.shape[1], good_centroids.
            shape[2]));

        for n_cluster in range(0, good_centroids.shape[1]):
            centroids = all_centroids[:, n_cluster, :];

            k = 1;
            while k < n_byzan:
                # Compute mean all of centroids
                mean_centroids = np.mean(centroids, axis=0);
                # Delete centroid that has the largest difference from
                    all_centroids
                distances = np.zeros((centroids.shape[0], ));

                for idx in range(0, centroids.shape[0]):
                    norm = np.linalg.norm(mean_centroids - centroids[idx]);
                    distances[idx] = norm;

                # The largest
                largest = distances.argmax();
                centroids = np.delete(centroids, largest, axis=0);

                k = k + 1;

            agregated_centroids[n_cluster] = np.mean(centroids, axis=0);

        return agregated_centroids;
```

Lien vers le code source : https://github.com/fokoa/byzantine_kmeans-msc_thesis