Large Scale Machine Learning

Project 2 : Large Scale Fast Gradient Descent Implementation with Spark/Scala
Lab session (Diamonds) : Large scale linear regression in Spark/Scala

AZOULAY Mikael
ATSOU Komi Bi-Ayéfo
BOUMAZA Khaoula
MISLAH Aymeric

May 2020

Promotion : M2 MIAGE 2019-2020

Tutor : Colazzo Dario

Introduction

Le sujet du projet choisi repose sur l'implémentation et la comparaison des différentes extensions des méthodes de descente de gradient dans un environnement distribué avec Spark. Il s'agit d'un algorithme de minimisation de la perte en calculant les gradients de la perte pour les paramètres du modèle, en fonction des données d'apprentissage. L'algorithme ajuste de manière itérative les paramètres afin de trouver progressivement la meilleure combinaison de pondérations et le biais pour minimiser la perte.

Le code est écrit en Scala avec Spark et nous avons réalisé différentes expérimentations afin de faire les comparaisons de performances entre les différentes variantes de la GD et pour plusieurs types de dataset (RDD, DataFrame, Dataset)..

I. Parallélisation des Gradient Descent

1. Parallel Gradient Descent : Algorithmie et Implémentation

L'objectif est d'implémenter ces différentes versions de Gradient Descent en parallèle afin de tirer parti au maximum de l'environnement Spark.

Dans ce papier : *Parallelized Stochastic Gradient Descent*[1] on nous explique comment paralléliser une Stochastic Gradient Descent. Nous avons implémenté l'algorithme ci-dessous et nous l'avons ensuite décliné pour les différentes versions de Gradient Descent.

Algorithm 3 SimuParallelSGD(Examples $\{c^1, \dots c^m\}$, Learning Rate η, Machines k)

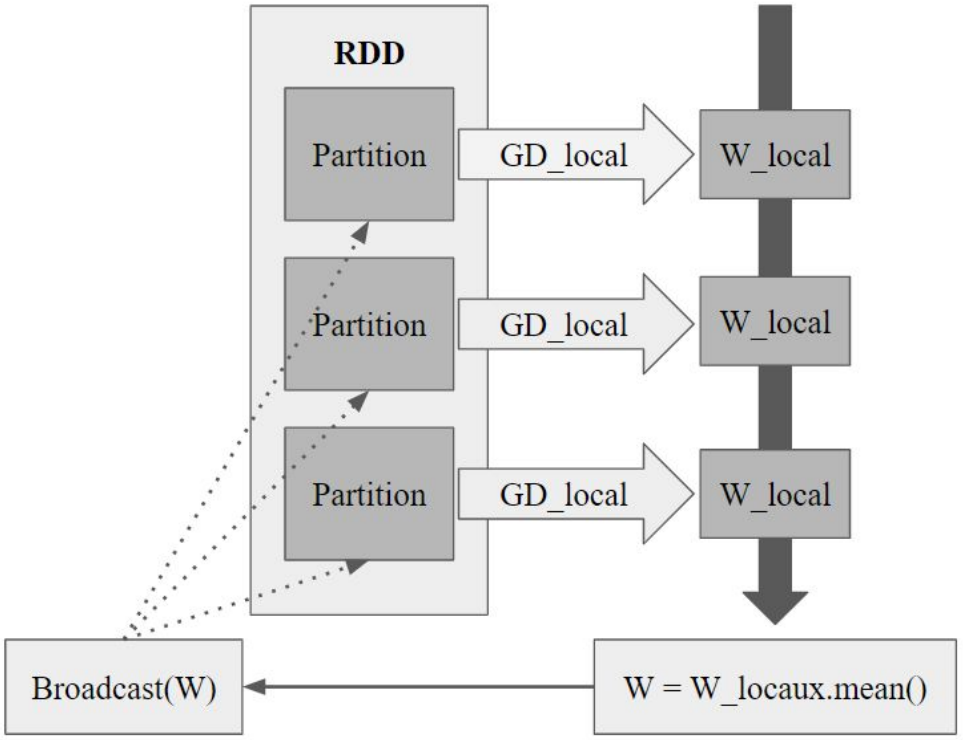
Define $T = \lfloor m/k \rfloor$
Randomly partition the examples, giving T examples to each machine.
for all $i \in \{1, \dots k\}$ **parallel do**
  Randomly shuffle the data on machine i.
  Initialize $w_{i,0} = 0$.
  for all $t \in \{1, \dots T\}$: **do**
    Get the tth example on the ith machine (this machine), $c^{i,t}$
    $w_{i,t} \leftarrow w_{i,t-1} - \eta \partial_w c^i(w_{i,t-1})$
  end for
end for
Aggregate from all computers $v = \frac{1}{k} \sum_{i=1}^{k} w_{i,t}$ and **return** v.

Chaque machine k effectue une SGD locale. On moyenne ensuite les k vecteurs W issus de ces k machines afin d'obtenir notre W final.

Pour implémenter cet algorithme en Spark nous avons considéré chaque machine k comme une partition de notre RDD/DataFrame/DataSet. On effectue une GD locale sur chaque partition k. On obtient donc k vecteurs *W_local*. Ces vecteurs sont moyennés en un vecteur *W_global* qui est diffusé (broadcast) aux k partitions qui effectuent à nouveau une GD locale

[1] http://martin.zinkevich.org/publications/nips2010.pdf

mais en partant de ce nouveau *W_global*. On recommence cette opération le nombre d'itérations voulues que l'on note epochs. Chaque partition k effectue un nombre d'itérations de GD locales afin d'obtenir son W_local, on note ce nombre d'itérations epochsLocal. On synthétise le fonctionnement de notre implémentation dans le schéma ci-dessous.



Nous avons adapté nos implémentations aux trois types de données : RDD, DataFrame et Dataset. Voici la liste des fonctions scala effectuant ces GD en local i.e. sur une partition :

MBGD_local : effectue une mini batch gradient descent (MBGD) en local

MOM_MBGD_local : effectue une MBGD avec momentum en local

ADA_MBGD_local : effectue une MBGD avec Adagrad en local

Voici la liste des fonctions scala effectuant ces GD en parallèle en s'appuyant sur les RDD :

MBGD_parallel_RDD

MOM_MBGD_parallel_RDD

ADA_MBGD_parallel_RDD

Voici la liste des fonctions scala effectuant ces GD en parallèle en s'appuyant sur les DataFrame :

MBGD_parallel_DF

MOM_MBGD_parallel_DF

ADA_MBGD_parallel_DF

Voici la liste des fonctions scala effectuant ces GD en parallèle en s'appuyant sur les Dataset:

MBGD_parallel_DS

MOM_MBGD_parallel_DS

ADA_MBGD_parallel_DS

2. Expérimentations à l'aide de Grid Search : Types et axes d'analyses

Afin de réaliser les expérimentations pour pouvoir faire des analyses et comparaisons d'une implémentation par rapport à l'autre et d'une variante de GD par rapport à l'autre, nous avons réalisé une fonction qui permet de réaliser plusieurs exécutions en fonction de paramètres fournis en entrée. Nos analyses sont de deux types en fonction de la nature de la variable d'intérêt. Il s'agit du temps d'exécution et du score des prédictions. Pour chaque type d'analyse, plusieurs axes d'analyses sont possibles en fonction de la situation analysée :

- Taille de dataset
- Taille de batch ou batch size (batch, mini-batch, stochastique)
- Nombres d'itérations pour converger
- Type de dataset (RDD, DataFrame, Dataset)
- Variante de Gradient Descent (Standard/Basique, Momentum, Adagrad)

Pour mesurer le temps de traitement, plusieurs méthodes sont possibles. Afin de pallier les variations qui pourraient arriver en fonction de l'état de la machine et de la JVM au moment où on fait les mesures, on peut faire plusieurs exécutions et faire des moyennes. Une méthode plus adaptée consiste à utiliser un outil de profiling comme Scalameter. Nous l'avons utilisé car il permet de réaliser la mesure lorsque la JVM est dans un état steady (à plein régime). Ce qui permet de réduire les fluctuations dans les mesures. Scalameter réalise plusieurs exécutions de la portion de programme dont on veut mesurer le temps jusqu'à atteindre l'état voulu et ensuite il fait la mesure. Dans notre cas, cela rallongeait considérablement les temps d'exécution. Nous avons donc fait des mesures simples en maîtrisant les fluctuations, par exemple en faisant des moyennes.

Nous avons fait un grid search pour les tests avec le dataset généré et un autre grid search pour le dataset minier. Les paramètres étant définis à l'appel du dataset, le grid search génère un fichier CSV avec les données issues de l'exécution.

En guise d'illustration, voici ci-après une capture des paramètres à fournir à la fonction de gridsearch pour le cas de la génération d'un problème linéaire et pour le dataset miniers, ainsi qu'un exemple d'utilisation avec les résultats de l'exécution :

Avec génération du problème linéaire

Cmd 9

```
def gridSearchGeneriqueDataSetGenere(tailleMaxDatasetAGenerer: Int,
                                     listePartitions: Array[Int],
                                     listeTypeDataset: Array[Int], // RDD = 0, DF = 1, DS = 2
                                     listeDataSize: Array[Int],
                                     listeBatchSize: Array[Int],
                                     listeVariantesGD: Array[Int], // Standard = 0, Momentum = 1, Adagrad = 2
                                     listeLocalParallele: Array[Int], // 0 = local, 1 = parralèle
                                     listeEpochsGlobal: Array[Int],
                                     listeEpochsLocal: Array[Int],
                                     pathCSVExport : String): DataFrame = {

    var lignesCVS = ArrayBuffer[Row]()
    var mapLigneCSV = Map[String, Any]()

```

Cmd 10

```
gridSearchGeneriqueDataSetGenere(
      listePartitions = Array(4, 8),
      listeBatchSize = Array(1, 32),
      listeDataSize = Array(10000),
      listeEpochsGlobal = Array(1),
      listeEpochsLocal = Array(1),
      listeLocalParallele = Array(1),
      listeTypeDataset = Array(0, 1, 2),
      listeVariantesGD = Array(0, 1, 2),
      tailleMaxDatasetAGenerer = 10000,
      pathCSVExport = "gridDSG.csv")
```

res1: org.apache.spark.sql.DataFrame = [varianteGD: string, typeDataset: string ... 8 more fields]

```
###### datasize = 10000
#### Partitions number = 4
## Epoch globale = 1
Epoch locale = 1
Batch size = 1
Batch size = 32
#### Partitions number = 8
## Epoch globale = 1
Epoch locale = 1
Batch size = 1
Batch size = 32
```

varianteGD	typeDataset	localParallele	dataSize	batchSize	partitionsNumber	epochGlobal	epochLocal	score	tempsExecutionMs
Standard	RDD	Parallele	10000	1	4	1	1	0.9999980315224476	0.635583643
Standard	DataFrame	Parallele	10000	1	4	1	1	0.9999980575869931	1.089874481
Standard	Dataset	Parallele	10000	1	4	1	1	0.9999979875484549	1.217629209
Momentum	RDD	Parallele	10000	1	4	1	1	0.9999977809578171	0.623686011
Momentum	DataFrame	Parallele	10000	1	4	1	1	0.9999981929597752	1.284762968
Momentum	Dataset	Parallele	10000	1	4	1	1	0.9999952030038145	1.348695631
Adagrad	RDD	Parallele	10000	1	4	1	1	0.999998000079918	0.602191728
Adagrad	DataFrame	Parallele	10000	1	4	1	1	0.9999996550900804	0.825092061
Adagrad	Dataset	Parallele	10000	1	4	1	1	0.9999989502350672	1.420068792
Standard	RDD	Parallele	10000	32	4	1	1	0.9669253119950633	0.174954117
Standard	DataFrame	Parallele	10000	32	4	1	1	0.9712165428673328	0.475257121
Standard	Dataset	Parallele	10000	32	4	1	1	0.9709985903762965	0.844618141
Momentum	RDD	Parallele	10000	32	4	1	1	0.9998783602667051	0.2974389
Momentum	DataFrame	Parallele	10000	32	4	1	1	0.9998185589239107	0.634267192
Momentum	Dataset	Parallele	10000	32	4	1	1	0.999921947227914	0.883345457
Adagrad	RDD	Parallele	10000	32	4	1	1	0.9999871362172119	0.36274842
Adagrad	DataFrame	Parallele	10000	32	4	1	1	0.9999891185295897	0.775268246
Adagrad	Dataset	Parallele	10000	32	4	1	1	0.9999906370075831	1.200860118
Standard	RDD	Parallele	10000	1	8	1	1	0.999998029660163	0.24874388
Standard	DataFrame	Parallele	10000	1	8	1	1	0.9999979846515752	0.607361746

only showing top 20 rows

Command took 1.17 minutes -- by marc.atsou@gmail.com at 25/05/2020 à 18:45:09 on My Cluster

Pour le dataset miniers:

md 11

```
def gridSearchGeneriqueMinierDataSet(pathDatasetTrain: String = "/FileStore/tables/dtrain.csv",
                                     pathDatasetTest: String = "/FileStore/tables/dtest.csv",
                                     listePartitions: Array[Int],
                                     listeTypeDataset: Array[Int], // RDD = 0, DF = 1, DS = 2
                                     listeDataSize: Array[Int],
                                     listeBatchSize: Array[Int],
                                     listeVariantesGD: Array[Int], // Standard = 0, Momentum = 1, Adagrad = 2
                                     listeLocalParallele: Array[Int], // 0 = local, 1 = parralèle
                                     listeEpochsGlobal: Array[Int],
                                     listeEpochsLocal: Array[Int],
                                     pathCVSExport : String
                                    ): DataFrame = {

    var lignesCVS = ArrayBuffer[Row]()
    var mapLigneCSV = Map[String, Any]()

```

Cmd 12

```
1   gridSearchGeneriqueMinierDataSet(
2     listePartitions = Array(4),
3     listeBatchSize = Array(32),
4     listeDataSize = Array(1000),
5     listeEpochsGlobal = Array(1),
6     listeEpochsLocal = Array(1),
7     listeLocalParallele = Array(1),
8     listeTypeDataset = Array(0, 1, 2),
9     listeVariantesGD = Array(0, 1, 2),
10    pathCVSExport = "gridM.csv")
```

▸ (52) Spark Jobs

▸ res7: org.apache.spark.sql.DataFrame = [varianteGD: string, typeDataset: string ... 8 more fields]

```
###### datasize = 1000
#### Partitions number = 4
## Epoch globale = 1
Epoch locale = 1
Batch size = 32
```

varianteGD	typeDataset	localParallele	dataSize	batchSize	partitionsNumber	epochGlobal	epochLocal	score	tempsExecutionMs
Standard	RDD	Parallele	1000	32	4	1	1	0.11811734173625299	13.912633286
Standard	DataFrame	Parallele	1000	32	4	1	1	0.10356113672178446	16.566223843
Standard	Dataset	Parallele	1000	32	4	1	1	0.10892702172058832	17.160120324
Momentum	RDD	Parallele	1000	32	4	1	1	0.5067123634201567	27.334261092
Momentum	DataFrame	Parallele	1000	32	4	1	1	0.5273436583320823	30.116399769
Momentum	Dataset	Parallele	1000	32	4	1	1	0.5305016323449121	33.730385207
Adagrad	RDD	Parallele	1000	32	4	1	1	0.13079476563810166	27.602699672
Adagrad	DataFrame	Parallele	1000	32	4	1	1	0.16058261227611537	32.197209106
Adagrad	Dataset	Parallele	1000	32	4	1	1	0.21668176674775652	34.106235592

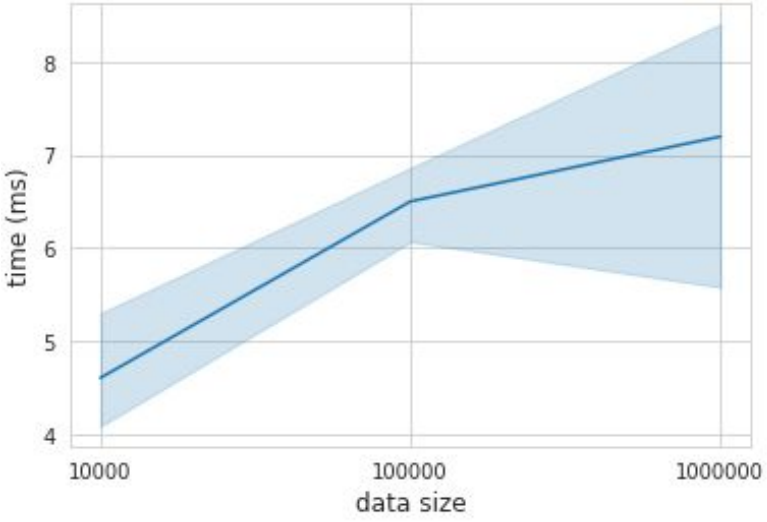
3. Performances de la parallélisation et comparaisons RDD, DataFrame et Dataset

Avant de mettre en œuvre nos travaux sur un vrai dataset (voir prochaine partie), nous avons réalisé des tests en implémentant des fonctions scala générant des problèmes linéaires génériques aléatoires de taille choisie : *generateRDD*, *generateDataFrame*, *generateDataSet.*

Elles génèrent respectivement un RDD, un DataFrame ou un DataSet de taille et de partitionnement choisi.

Pour se rendre compte des différences de performances de nos différentes implémentations nous avons choisi de mesurer leur vitesse d'exécution pour une itération de GD pour des problèmes génériques de différentes tailles. Pour cela on utilise le grid search décrit plus haut afin de lancer nos expériences en croisant les différents paramètres.

Ci dessous le graphique donne la vitesse d'exécution moyenne d'une itération de GD parallèle

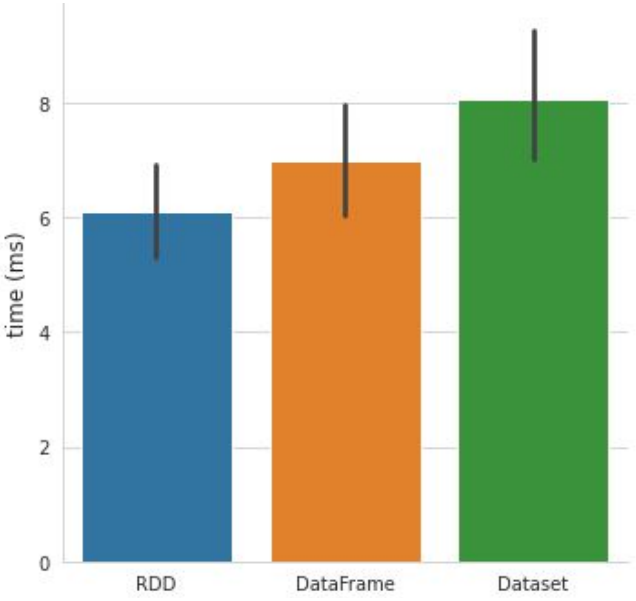


La ligne en gras bleue donne le temps moyen d'une itération, la zone en bleue représente les temps maximum et minimum rencontrés lors de nos expérimentations. En x le nombre de lignes du dataset de 10 000 à 1 000 000, en y le temps d'exécution d'une itération de GD.

Nos implémentations offrent de bons résultats. On multiplie par 10 la taille du dataset deux fois et le temps d'exécution évolue de 50% la première fois et de 15% la seconde fois.

Pour chacune des variantes de GD parallèle nous disposons de 3 implémentations scala : une basée sur les RDDs, l'autre sur les DataFrames et la dernière sur les Datasets. Comme fait plus haut nous allons lancé un grid search et moyenné les résultats afin d'obtenir un temps d'exécution moyen d'une itération de GD (en y) selon le type d'implémentation (en x).

Le barplot ci dessous donne ces résultats ainsi que les bornes inférieures et supérieures de nos expériences (lignes noires).



Notre implémentation RDD est plus rapide puis on retrouve celle basée sur DataFrame et enfin celle basée Dataset. Les DF et DS étant réputés plus rapides, comment peut-on expliquer ce constat ?

Nous pouvons émettre quelques hypothèses basées sur nos expériences et sur notre connaissance de Spark.

- D'abord les GD font appel à des calculs d'algèbre linéaire pour lesquels les DF et DS ne sont pas calibrés. On fait généralement appel à ces type de données pour des opérations relationnelles de jointures et d'agrégations ; ce qui n'est pas le cas ici. Les DF et les DS apparaissent peut-être dans le cas de nos implémentations comme une complexification inutile du problème.
- La deuxième hypothèse peut provenir de l'architecture d'exécution. Etant donné que nos exécutions se font sur une seule machine qui simule le parallélisme en utilisant les processeurs, ce type d'architecture peut être à l'origine de certaines observations. Par exemple l'utilisation de mapPartitions sur une architecture mono-machine ne permet pas de voir les différences de performance qu'on pourrait avoir sur une vraie architecture distribuée avec plusieurs machines.

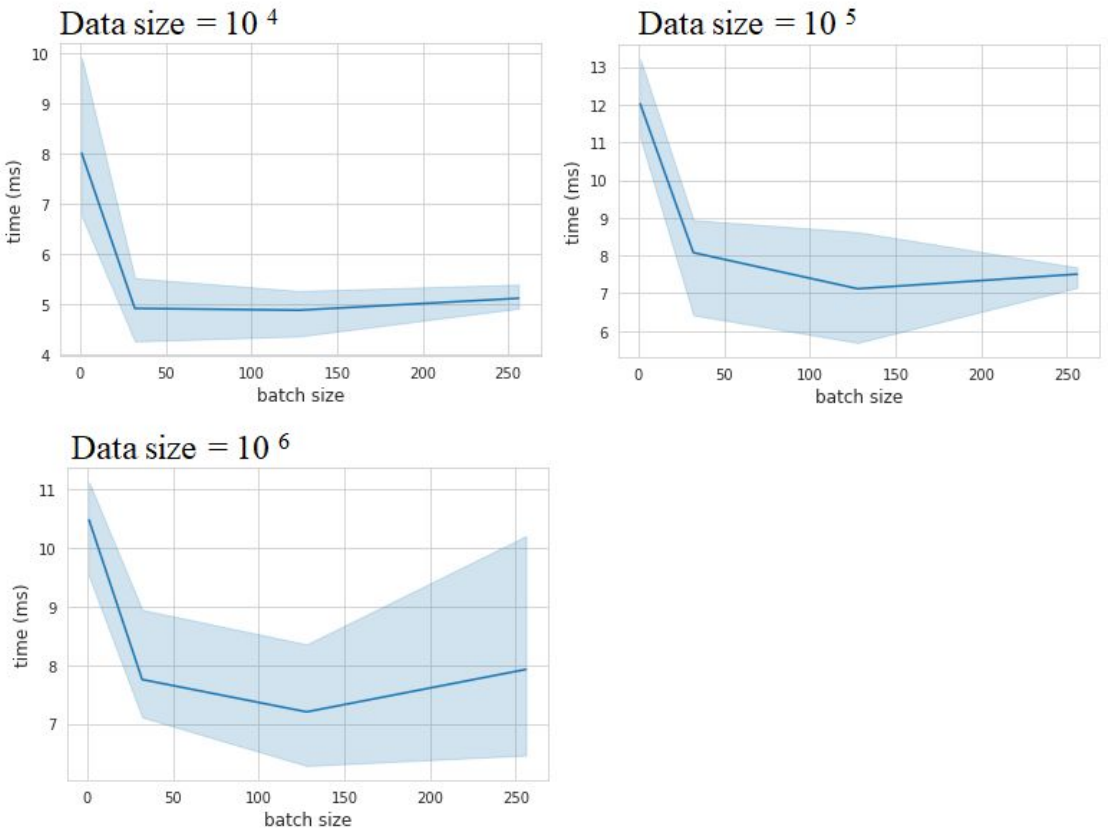
II. Comparaisons Gradient Descent basiques et Fast Gradient Descent

Dans cette partie nous allons comparer les différentes implémentations des variantes de GD que nous avons choisies. Plus précisément nous étudierons l'impact de la taille du batch (batch, mini-batch, SGD) sur la vitesse d'exécution et nous comparerons ensuite les GD basiques avec les Fast GD au niveau de leur vitesse de convergence.

1. Gradient Descent basiques et batching (batch, mini-batch, SGD)

Dans les GD basiques on compte la batch GD, la mini batch GD ainsi que la stochastic GD. Ces trois GD ont pour unique différence la taille de batch c'est à dire le nombre d'exemples piochés dans les données à chaque étape afin de mettre à jour les poids.

Là où la batch gradient descent se met à jour sur tout le dataset à la fois, la mini batch utilise des petits lots de données et la stochastic une unique instance. La question de la taille de batch est cruciale et dépend de la taille du dataset. En effet une batch gradient descent est rarement envisageable sur un problème volumineux. La batch size a ainsi un impact fort sur la vitesse d'exécution et nous avons utilisé notre grid search afin d'expérimenter différentes tailles de batch pour différentes tailles de dataset. Les résultats sont décrits dans les graphiques ci-dessous avec la data size en nombre de lignes.



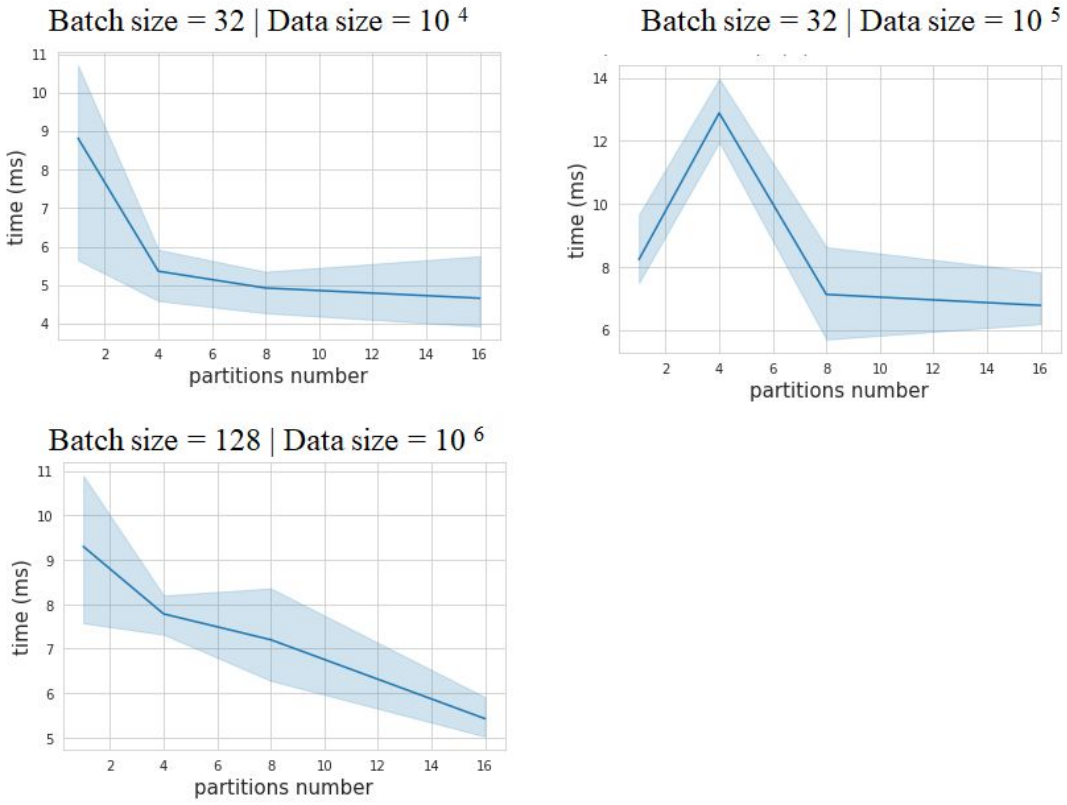
On remarque que dans les trois cas les courbes repartent à la hausse lorsque la batch size dépasse un certain seuil. Il est déjà clair qu'en termes de vitesse d'exécution la batch gradient descent est à éviter pour des datasets volumineux.

Dans le cadre de nos expériences, une batch size trop faible est également pénalisant, cela exclue donc la stochastic gradient descent et sa taille de batch égale à 1. Il vaudra mieux privilégier un mini batch d'une taille optimale proportionnelle à la taille du dataset mais aussi dépendante de l'architecture de calcul de notre environnement d'exécution. Dans nos cas pour des datasets larges et très larges, une taille de batch de 128 est un bon choix. Pour les problèmes plus petits, une taille de 32 est souvent efficace.

2. Le partitionnement

Le partitionnement détermine en combien de partitions sera découpé notre jeu de données soit le nombre de parallélisations à effectuer.

Dans le cas de nos implémentations, le nombre de partitions de nos données a un fort impact sur la vitesse d'exécution. Un nombre de partitions trop grand ou trop petit augmente le temps d'exécution. Ces paramètres dépendant fortement de l'architecture de calcul des machines exécutant le programme ainsi que de la dimension du problème à résoudre, nous utiliserons notre gridsearch afin de trouver le partitionnement idéal pour notre problème générique sur notre environnement d'exécution.



Dans les graphiques ci-dessus, on a, pour trois tailles de dataset différentes, retenu le batching size idéal et on a ensuite expérimenté plusieurs partitionnements. On remarque que la tendance est la suivante : alors même que ces expérimentations ont été faites sur une machine en local, plus le partitionnement est élevé plus on réduit le temps d'exécution jusqu'à atteindre le minimum lorsqu'on est à 16 partitions. Combiné à un bon choix de taille de batch on peut réussir à réduire considérablement le temps d'exécution d'une itération.

3. Performances des Fast GD par rapport à la version basique

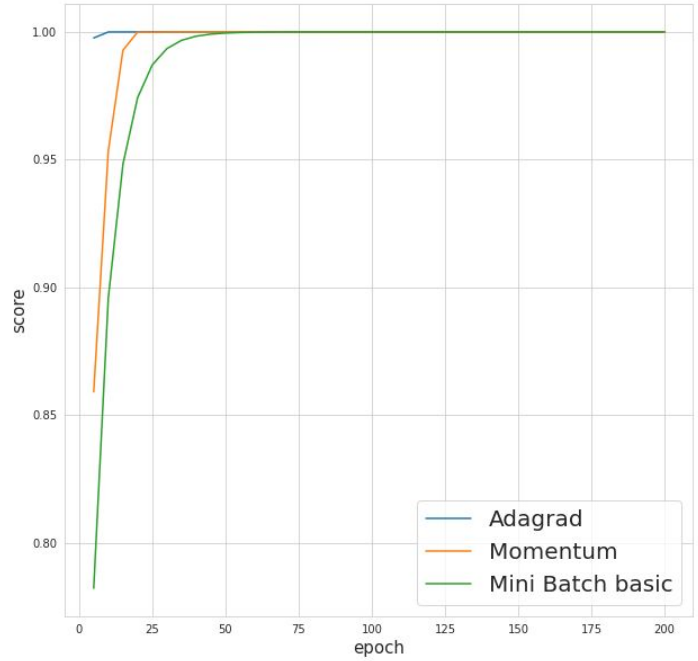
Nous allons à présent comparer les implémentations basiques de GD aux Fast SGD.

Les basiques GD comprennent les GD avec mini batch de différentes tailles allant de 1 (stochastic gradient descent) à la taille du dataset (batch gradient descent).

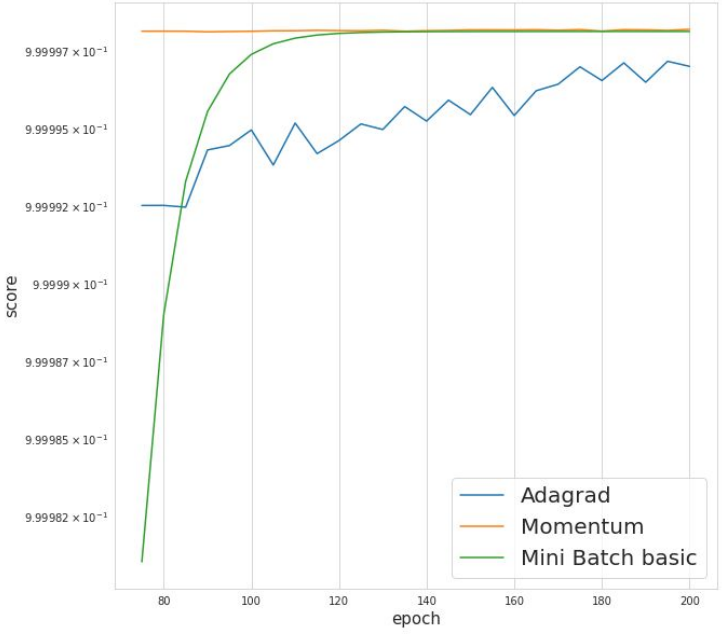
En Fast SGD nous avons choisi d'implémenter ces mini batch gradient descent avec Adagrad ou Momentum comme le demande le sujet.

Nous comparons, pour une taille de batch fixée arbitrairement et un problème généré arbitrairement la vitesse de convergence de ces différentes GD.

Le graphique ci-dessous montre l'évolution du r2_score des différentes variantes de GD au cours des 100 premières itérations :



Le graphique ci-dessous montre l'évolution du r2_score des différentes variantes de GD au cours des 100 dernières itérations :



On remarque que la MBGD avec Adagrad obtient en moins de 5 itérations un score de plus de 0.99. Il faut 25 itérations pour que MBGD avec Momentum la rejoigne et 50 pour la MBGD basique.

Après 80 itérations la Adagrad est finalement dépassée par la MBGD basique et la Momentum.

Interprétation : Les Fast GD offrent une convergence beaucoup plus rapide que la MBGD classique. Cependant sur le long terme la MBGD offre une convergence certe plus lente mais plus sûre. Le choix de la variante de GD se fera donc selon les impératifs définis pour résoudre le problème. Si l'on dispose d'un dataset très volumineux et/ou que l'on exige des convergences pas trop longue on privilégiera les Fast GD qui offriront de très bons scores en peu d'itérations. En revanche si la contrainte de temps importe peu où que notre dataset est de petit taille le choix d'une MBGD classique est un choix sur.

III. Prédiction de la qualité de minerais

Dans cette partie nous avons voulu mesurer les performances de nos différentes Gradient Descent dans la résolution d'un problème concret de Machine Learning. Pour se faire nous avons choisi un dataset large qui nous permettrait de tester la scalabilité de nos implémentations. Nous allons essayer de prédire au mieux la qualité d'une extraction de minerais.

1. Description du dataset

Les données sont issues d'un processus d'extraction de minerais et plus précisément de la partie la plus importante de cette extraction, la flottation. Cette technique de séparation est fondée sur les propriétés de surface et sur les propriétés d'hydrophobie et d'hydrophilie des phases minérales et a lieu dans une cellule de flottation. Les données ont été trouvées sur **Kaggle**[2] et donne pour chaque heure de flottation les concentrations en impureté dans le mélange. Avec ces données nous pouvons tenter, à l'aide de machine learning, de prédire la quantité de silice (impureté) dans le concentré de minerai, et ainsi aider les ingénieurs, en leur donnant des informations précoces pour prendre des mesures.

Le dataset compte 24 colonnes (23 features et 1 label) toutes numériques (sauf la première) et près de 740 000 enregistrements pour un total de 175 MB.

La première colonne indique la plage d'heures et de dates (de mars 2017 à septembre 2017). Certaines colonnes ont été échantillonnées toutes les 20 secondes. D'autres ont été échantillonnées sur une base horaire.

Les deuxième et troisième colonnes sont des mesures de la qualité de la pâte de minerai de fer juste avant son introduction dans l'usine de flottation. La colonne 4 jusqu'à la colonne 8 sont

[2] https://www.kaggle.com/edumagalhaes/quality-prediction-in-a-mining-process

les variables les plus importantes qui ont un impact sur la qualité du minerai à la fin du processus. De la colonne 9 à la colonne 22, nous pouvons voir les données du processus (niveau et débit d'air à l'intérieur des colonnes de flottation), qui ont également un impact sur la qualité du minerai. Les deux dernières colonnes sont la mesure finale de la qualité de la pâte de minerai de fer du laboratoire.

L'objectif est de prédire la dernière colonne, qui est le % de silice (impureté) dans le concentré de minerai de fer.

2. Préparation des données

Avant de se lancer dans la résolution du problème on a dû appliquer toutes sortes de transformations sur nos données afin de réaliser les Gradient Descent. Cette préparation des données est aussi valable pour le dataset diamonds

Chargement des données

On charge d'abord le .csv dans un dataframe et on le répartit en X partitions voulues.

Nettoyage des données

On supprime les colonnes inutiles et on remplace les valeurs null par la moyenne. Pour se faire on utilise un *Imputer* que l'on retrouve dans la librairie *Spark ML*.

Indexation des features catégoriques

Traitement sur les colonnes catégoriques du dataset *diamonds* à l'aide de *StringIndexer* de *Spark ML.* On passe d'une catégorie string à une valeure numérique.

One hot encoding des features catégoriques

Traitement sur les colonnes catégoriques du dataset *diamonds* à l'aide de *OneHotEndcoderE* de *Spark ML.* On passe nos catégories indexées à un vecteur.

Assemblage des features

On utilise *VectorAssembler* de *Spark ML* pour regrouper nos features en 1 seul et même *SparseVector*.

Division des données

On utilise encore Spark ML pour regrouper nos features dans une colonne et nos labels dans une autre. On divise ensuite notre dataset en un *training set* (80% des instances) sur lequel nos GD s'entraîneront et un *test set* (20%) sur lequel ils feront leur prédictions.

Normalisation des données

Pour accélérer la convergence des GD il faut normaliser nos données. On utilise un *StandardScaler* de Spark ML qui transforme nos features, normalisant chaque entité pour avoir un écart-type unitaire et / ou une moyenne nulle.

Au niveau de notre implémentation Scala, tous ces traitements se trouvent dans la fonction *preprocessMinier* qui prend en entrée le pathfile vers le .csv, le nombre de partitions voulues de notre DataFrame et le nom de la colonne contenant les labels à prédire. En sortie la fonction renvoie deux dataframes (train set et test set) traités de chacun deux colonnes. La colonne *'labels'* avec des valeurs de type Double et la colonne 'features' avec des valeurs de type Array[Double], un tableau de 23 valeurs numériques.

Pour le dataset *diamonds.csv* la fonction scala se nomme *preprocessDiamonds* et renvoie cette fois 4 DataFrames. Deux compatibles (train set et test set) avec les régressions spark ml et deux autres (train set et test set) compatibles avec nos GD c'est à dire avec les features dans un Array.

3. Calcul de l'erreur et du score

Pour établir des prédictions il suffit de faire le produit scalaire des features du problème par le vecteur de poids W émis par nos Gradient Descent. On obtient ainsi pour chaque instance x_i de notre problème linéaire, sa valeur prédite associée y_pred_i.

On cherche à minimiser la somme des erreurs entre les valeurs prédites et les valeurs réelles de nos données. Pour calculer cette erreur on implémente la fonction *mse* qui calcule la mean squared error de notre problème définit comme suit :

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Une autre façon de mesurer les performances de nos GD est de faire appelle au R-squared score. Ce score mesure les performances d'un modèle de régression linéaire. Ce score est calculé de la façon suivante et le calcul est implémenté dans la fonction *r2_score*:

1/ On calcule la moyenne des y notée :

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

2/ On calcule le « total sum of squares » (proportionnel à la variance des données) :

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

3/ On calcule le « residual sum of squares » (la somme des erreurs au carré) :

$$SS_{\text{res}} = \sum_i (y_i - f_i)^2$$

4/ Enfin on calcule le R2 score :

$$R^2 \equiv 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

Dans le meilleur des cas les valeurs prédites sont exactement les valeurs réelles, autrement dit SS res = 0 et donc R2 = 1. Un score de 0 signifie que les prédictions valent toutes la moyenne, le modèle n'a aucune influence. Un score négatif signifie que le modèle est mauvais et fait pire que le hasard.

4. Résolution du problème

Une fois nos données traitées avec la fonction *preprocess* il nous faut résoudre notre problème de régression à l'aide de nos Gradient Descent.

On entraîne nos Gradient Descent sur le train set et on teste leur performance sur le test set. Les performances sont mesurées à l'aide des fonctions *r2_score* et *mse* décrites plus haut.

On se fixe un nombre élevé d'itérations pour l'entraînement et on choisira comme solveur la Gradient Descent ayant le score le plus élevé à leur terme.

Pour 100 itérations (10 *epochs* x 10 *epochsLocal*) soit 5 minutes d'exécution par GD on obtient les scores suivants :

	MBGD	MOM MBGD	ADA MBGD
error (mse)	0.48066	0.45149	0.47063
score (r2_score)	0.62112	0.64623	0.62903

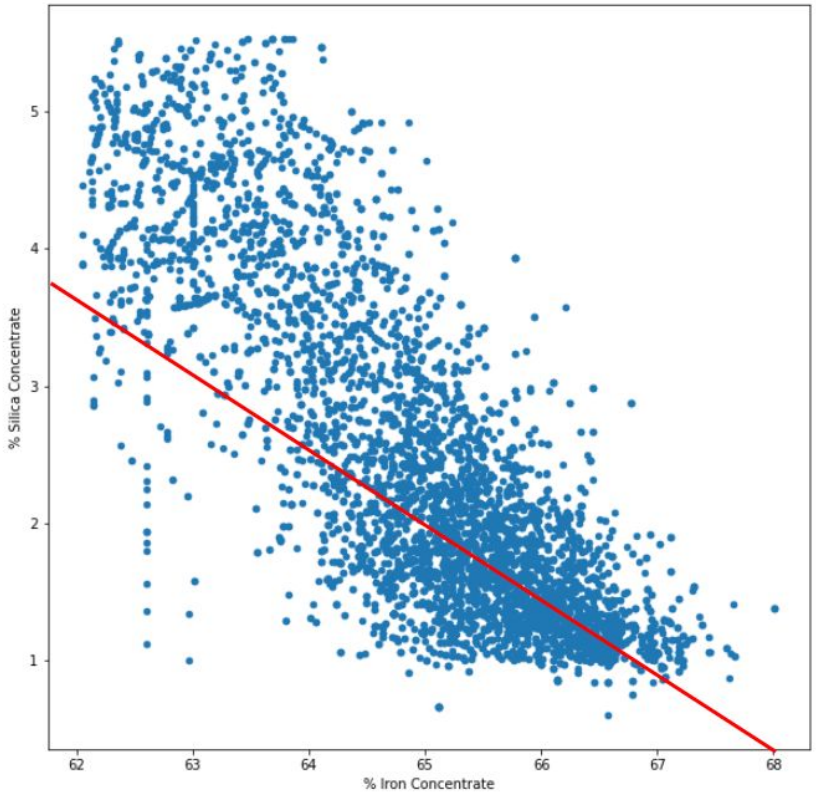
On choisit donc les prédictions réalisées par notre mini batch gradient descent avec un momentum égal à 0.9. Ce modèle se trompe d'en moyenne 0.45 en termes de concentration de silicate (impuretées) dans notre minerais de fer. Sachant que cette concentration est d'en moyenne 2.7. Notre modèle offre donc des prédictions utiles aux ingénieurs lors de l'extraction minière.

Nous allons visualiser en partie nos prédictions (sur 1% du dataset).

Pour cela on se place en x au niveau de notre feature le plus important : % Iron Concentrate.

En y au niveau des labels à prédire : % Silicate Concentrate

En rouge on affiche la droite de régression associée à ce feature ayant pour coefficient directeur le w_i associé.



Pour des concentrations en silice inférieure à 3% notre modèle linéaire épouse au mieux le nuage de points.

En revanche, notre modèle rencontrera des difficultés lorsqu'il s'agira de prédire des concentrations en silice supérieures à 3%. Il s'agit là des limites d'un modèle linéaire. On imagine que des modèles polynomiaux seraient peut-être plus efficaces dans ce cas.

TP Diamonds : Régression linéaire en Spark/Scala

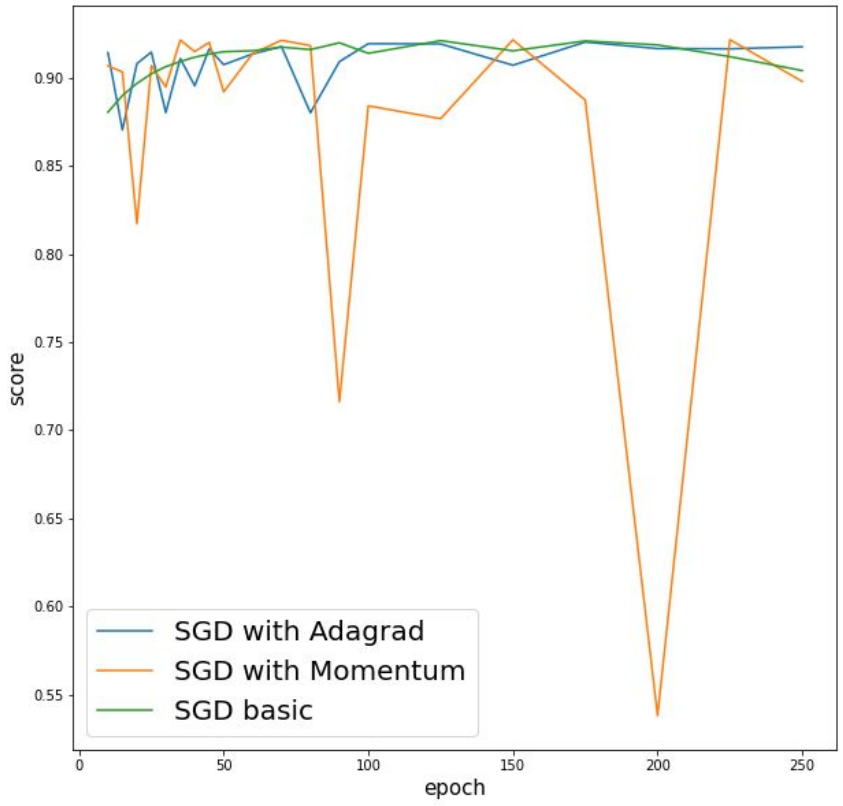
Comme demandé, nous avons poursuivi le travail à faire dans le cadre du TP Diamonds. Le notebook ainsi que les commentaires sont ajoutés sous forme de pièce jointe à ce document.

Le travail effectué ressemble à celui réalisé dans la partie *prédiction de la qualité de minerais* mais dans ce cas nous nous sommes penchés sur le dataset *diamonds* décrits dans le cours.

La différence avec le travail réalisé plus haut est que dans ce cas nous avons confronté nos GD aux modèles de régression linéaire disponible sur Spark ML.

Les traitements sur les données sont là aussi légèrement différents et regroupés dans la fonction *preprocessDiamonds*.

Dans le graphique ci dessous on observe l'évolution du r2_score de nos 3 GD parallèles au cours de 250 itérations. Dans ce cas la taille de batch optimale et retenue est 1 on nos GD sont donc : la SGD basique, SGD avec Momemtum = 0.9 et SGD avec Adagrad.



Même constat que celui fait plus haut les Fast SGD convergent plus rapidement mais au final sur le long terme la SGD basique les rejoins en terme de score. A noter qu'au bout de 125 itérations la SGD basique converge et entre ensuite en surapprentissage. Il en 35 pour celle avec momentum et seulement 10 pour adagrad.

Dans le le tableau ci dessous sont on récapitule les meilleurs résultats en terme de r2_score de nos meilleurs SGD à ceux de spark ML. Nos modèles sont plus lents que ceux de Spark ML à converger mais fournissent des résultats équivalents. On peut également se contenter de 5 secondes d'exécution avec nos SGD pour avoir des résultats avoisinant 0.9 en r2_score.

	SGD	SGD MOM	SGD ADA	LR Spark ML
r2_score	0.921107	0.921662	0.920305	0.922537
time to converge (sec)	55.976617	46.160000	23.528104	2.18

Nos SGD peuvent fournir de très bons résultats quasiment aussi bon que ceux offerts par les régressions linéaires de Spark ML et en peu d'itérations.

Mais si on leur laisse plus de temps pour s'entraîner ils peuvent même dépasser les modèles de spark ML en terme de mse et de r2_score.

Ci-dessous le résultat du un récapitulatif de l'entraînement de nos SGD pour diamonds :

epoch	mbgdscore	mbgderror	mbgdtime	momscore	momerror	momtime	adascore	adaerror	adatime
10.0	0.880511	1.855628e+06	7.427229	0.906751	1.448127e+06	17.116573	0.914263	1.331462e+06	23.528104
15.0	0.889981	1.708561e+06	8.999423	0.903319	1.501421e+06	22.768015	0.870422	2.012308e+06	32.053560
20.0	0.896796	1.602721e+06	12.630520	0.817196	2.838893e+06	27.871358	0.908044	1.428047e+06	40.766532
25.0	0.902314	1.517025e+06	14.192480	0.906921	1.445489e+06	33.577840	0.914615	1.326006e+06	48.022942
30.0	0.906316	1.454880e+06	15.648391	0.894807	1.633613e+06	38.549004	0.880288	1.859096e+06	55.681534
35.0	0.909273	1.408967e+06	16.615323	0.921477	1.219437e+06	46.160000	0.910918	1.383415e+06	61.603712
40.0	0.911689	1.371434e+06	18.519783	0.914866	1.322110e+06	50.546395	0.895464	1.623404e+06	72.428163
45.0	0.913458	1.343973e+06	21.692523	0.920107	1.240715e+06	54.948285	0.916415	1.298046e+06	76.857609
50.0	0.914817	1.322858e+06	22.679659	0.891979	1.677532e+06	59.603645	0.907447	1.437325e+06	91.314297
60.0	0.915368	1.314314e+06	28.457252	0.913725	1.339826e+06	74.062693	0.913546	1.342595e+06	103.904899
70.0	0.917432	1.282252e+06	30.914934	0.921287	1.222386e+06	87.750807	0.917729	1.277639e+06	123.611579
80.0	0.916031	1.304015e+06	37.322208	0.918279	1.269106e+06	97.801928	0.880142	1.861358e+06	141.294816
90.0	0.919876	1.244306e+06	39.724335	0.716201	4.407306e+06	109.259816	0.909161	1.410697e+06	153.084550
100.0	0.913897	1.337156e+06	42.694328	0.884098	1.799920e+06	118.506469	0.919370	1.252161e+06	172.437864
125.0	0.921107	1.225180e+06	55.976617	0.876791	1.913393e+06	147.844804	0.919254	1.253963e+06	212.891329
150.0	0.915316	1.315115e+06	68.446360	0.921567	1.218039e+06	167.840212	0.907130	1.442245e+06	253.412706
175.0	0.920928	1.227957e+06	78.987356	0.887441	1.747998e+06	196.431155	0.920305	1.237635e+06	283.464748
200.0	0.918677	1.262925e+06	85.500301	0.537971	7.175165e+06	215.564938	0.916594	1.295261e+06	315.698334
225.0	0.912018	1.366328e+06	95.925772	0.921662	1.216570e+06	241.338549	0.916451	1.297486e+06	356.363699
250.0	0.904111	1.489124e+06	110.305227	0.897959	1.584658e+06	293.029541	0.917596	1.279707e+06	397.725166

Difficultés rencontrées

Ce travail serait moins enrichissant s'il ne comportait pas de défis et challenges à gérer. En effet, nous avons été confrontés à pas mal de difficultés. Heureusement, nous avons réussi à surmonter la plupart d'entre elles avec de la documentation complémentaire sur Internet. Un des défis majeurs auxquels nous étions confrontés consistait à comparer certains paramètres (nombre d'itérations, score, etc…) pour un temps limité d'exécution, avec l'utilisation des Futures, Threads et Promises. L'idée était de voir entre plusieurs variantes de GD laquelle convergeait plus vite pour un temps d'exécution donné. Finalement nous avons trouvé d'autres façons plus simples de réaliser cette analyse. Nous tenons à mentionner une des difficultés majeures qui tenait simplement dans le fait que nous ne disposions pas d'une vraie architecture distribuée. Même les clusters sur le cloud ne proposent qu'une machine worker dans leurs abonnements gratuits (Databricks, GCP).

Conclusion

Nous avons parié que ce projet serait bénéfique à plus d'un titre. Au moment où nous écrivons ces lignes, nous sommes convaincus que notre pari était justifié. En effet, la réalisation de ce projet nous a permis de monter en compétences sur plusieurs aspects du machine learning à grande échelle avec Spark. Il s'agit notamment des contraintes liées au choix des structures de données à utiliser, les détails théoriques des algorithmes qui peuvent avoir des influences non négligeables lorsqu'on passe à l'échelle, et les contraintes générales d'analyse et de visualisation de données mais dans un contexte big data. Enfin, nous avons beaucoup progressé dans l'utilisation d'un nouveau langage de programmation à savoir Scala.

Ce travail est certes perfectible. Plusieurs pistes d'amélioration sont possibles. D'abord en ce qui concerne les analyses, d'autres types d'analyses peuvent être effectuées surtout si on dispose d'une architecture vraiment parallèle. Par exemple, analyser les performances en termes de shuffles and sorts, en termes de mémoire RAM utilisée, en termes de données transférées via le réseau pendant les traitements. Ceci dit, nous sommes ouverts à toute proposition permettant de parfaire notre travail.

Annexes

- Notebook du projet : Voir pièce jointe

- Notebook du TP : Voir pièce jointe

- Quelques résultats des grid search exportés en CVS pour les visualisations

	A	B	C	D	E	F	G	H	I	J
1	varianteG	typeDatas	localParalle	dataSi	batchSi	partitionsNumb	epochGlob	epochLoc	score	tempsExecutionM
2	Standard	Array	Local	100	40	3	1	3	0.99999801	0.3076576
3	Momentum	Array	Local	100	40	3	1	3	0.999998037	0.172212
4	Standard	RDD	Parallele	100	40	3	1	3	0.999891019	0.2502924
5	Momentum	RDD	Parallele	100	40	3	1	3	0.999998063	0.4445718
6	Adagrad	RDD	Parallele	100	40	3	1	3	0.99999218	0.4520547
7	Standard	DataFrame	Parallele	100	40	3	1	3	0.99991303	0.6868014
8	Momentum	DataFrame	Parallele	100	40	3	1	3	0.99999803	0.7230233
9	Adagrad	DataFrame	Parallele	100	40	3	1	3	0.999992481	0.6786618
10	Standard	Dataset	Parallele	100	40	3	1	3	0.999899072	0.6503329
11	Momentum	Dataset	Parallele	100	40	3	1	3	0.999998052	0.8169766
12	Adagrad	Dataset	Parallele	100	40	3	1	3	0.999991097	1.0829525
13	Standard	Array	Local	100	60	3	1	3	0.999997979	0.0570397
14	Momentum	Array	Local	100	60	3	1	3	0.999998026	0.0882944
15	Standard	RDD	Parallele	100	60	3	1	3	0.998287731	0.1114459
16	Momentum	RDD	Parallele	100	60	3	1	3	0.999997915	0.2436851
17	Adagrad	RDD	Parallele	100	60	3	1	3	0.999990251	0.4106075
18	Standard	DataFrame	Parallele	100	60	3	1	3	0.998495752	0.1841731
19	Momentum	DataFrame	Parallele	100	60	3	1	3	0.999998051	0.3810408
20	Adagrad	DataFrame	Parallele	100	60	3	1	3	0.999991056	0.4208517
21	Standard	Dataset	Parallele	100	60	3	1	3	0.998265733	0.4530511
22	Momentum	Dataset	Parallele	100	60	3	1	3	0.99999797	0.7213543
23	Adagrad	Dataset	Parallele	100	60	3	1	3	0.999991624	0.6955637
24	Standard	Array	Local	100	100	3	1	3	0.999975206	0.0604456
25	Momentum	Array	Local	100	100	3	1	3	0.999998039	0.0810176
26	Standard	RDD	Parallele	100	100	3	1	3	0.984446958	0.1046227
27	Momentum	RDD	Parallele	100	100	3	1	3	0.999986372	0.4154473
28	Adagrad	RDD	Parallele	100	100	3	1	3	0.999988913	0.2428126
29	Standard	DataFrame	Parallele	100	100	3	1	3	0.986107942	0.202505
30	Momentum	DataFrame	Parallele	100	100	3	1	3	0.999989674	0.4431456
31	Adagrad	DataFrame	Parallele	100	100	3	1	3	0.999988818	0.2781409
32	Standard	Dataset	Parallele	100	100	3	1	3	0.984968018	0.2981937
33	Momentum	Dataset	Parallele	100	100	3	1	3	0.999991391	0.4113113

- Liste de quelques fonctions principales du Notebook

Génération de problèmes linéaires standards :

generateRDD

generateDataFrame

generateDataSet

Traitement des données :

preprocessMinier

preprocessDiamonds

preprocessEtExport

Gradient Descent :

MBGD_local

MOM_MBGD_local

ADA_MBGD_local

MBGD_parallel_RDD

MOM_MBGD_parallel_RDD

ADA_MBGD_parallel_RDD

MBGD_parallel_DF

MOM_MBGD_parallel_DF

ADA_MBGD_parallel_DF

MBGD_parallel_DS

MOM_MBGD_parallel_DS

ADA_MBGD_parallel_DS

Evaluation des modèles :

r2_score

mse

Grid Search :

gridSearchGeneriqueMinierDataSet

gridSearchGeneriqueDataSetGenere