MapReduce pour Statisticien

Résumé

Introduction rudimentaire aux fonctionnalités de MapReduce en utilisant la librairie `rmr2` *de RHadoop qui permet de simuler des gestions de fichiers HDFS sans Hadoop. Installation, illustration de l'écriture de fonctions Map et Reduce dans R, programmes élémentaires pour la régression et la classifications non supervisée ; limites de RHadoop.*

Organisation des tutoriels R.

- *Démarrer rapidement avec R*
- *Initiation à R*
- *Fonctions graphiques de R*
- *Programmation en R*
- *MapReduce pour le statisticien*
- *Apprentissage et données massives avec H2O*

1 introduction

1.1 Objectif

L'objectif de ce tuteuriel est de partiellement illustrer une introduction au traitement de données massives en montrant les difficultés engendrées par la gestion de fichiers trop gros pour être chargés en mémoire. L'environnement dominant *Hadoop* permet cette gestion en distribuant les algorithmes de calcul vers les données réparties mais génère de très fortes contraintes dues aux spécificité de la parallélisation dite *MapReduce*. Cette vignette propose donc un tuteuriel d'initiation aux fonctionnalités de MapReduce imposées par les structures de fichier (HDFS) concernées.

La première étape consiste à consulter la présentation sommaire du système de gestion de fichier Hadoop et des fonctionnalités MapReduce, avant d'exécuter les commandes ci-dessous. Il existe de très nombreuses technologies concurrentes pour traiter les données massives mais, pour éviter les nombreux problèmes de réseaux, serveurs, programmation... qu'elles soulèvent, le choix est ici fait de se limiter à une librairie (`rmr2`) de RHadoop. Ce n'est clairement pas la solution la plus efficace en terme de temps de calcul mais, basée sur R, elle permet d'aborder les principaux problèmes algorithmiques sans complications informatiques et ce, d'autant plus, qu'il est possible de simuler une gestion de fichiers HDFS sans avoir à installer Hadoop sur une machine, voire un cluster.

RHadoop est le produit d'appel *open source* de la société *Revolution Analytics* qui facture par ailleurs services et logiciels pour l'étude de données massives et propose un tuteuriel de RHadoop dont s'inspire largement ce document. Ce tutoriel met en avant les fortes contraintes de Map Reduce à l'origine de temps de calcul rédhibitoires pour des algorithmes itératifs. *Revolution Analytics* a d'ailleurs été récheté par Windows en début d'année 2015.

1.2 Installation

La librairie n'est pas accessible sur le CRAN mais dans un environnement collaboratif "git" ; y télécharger la dernière version de `rmr2`. Les autres librairies ne sont nécessaires que pour un environnement complet Hadoop dont l'installation n'est pas franchement du ressort du statisticien.

Le chargement de `rmr2` dépend évidemment d'autres librairies dont l'installation ne pose guère de problème à l'exception de `rJava` qui peut s'avérer délicate dans un environnement Windows. Un autre tuteuriel vise l'installation sous Linux.

Les exemples ci-dessous de traitement paraissent longs et très complexes au regard de l'extrême simplicité des exemples présentés. Néanmoins ces traitements sont adaptés aux données massives contrairement aux quelques lignes de R qui fournissent les mêmes résultats lorsque toutes les données tiennent en mémoire.

1.3 Fonctions

Consulter évidemment l'aide en ligne pour une description exhaustive, mais voici quelques fonctions qui manipulent des objets "big data" : `big.data.object` stockés dans un environnement (HDFS) Hadoop.

`keyval, values, keys` crée des paires (clef, valeur) ou extrait les valeurs (resp. clefs) de ces paires.

`to.dfs(kv, output)` où `kv` est une liste de (clef, valeur) ou un objet R et alors par défaut, la clef est `NULL` ; `output` est un fichier ou objet *big data* , par défaut un fichier temporaire.

`from.dfs(input)` transforme l'objet *big data* en entrée en un objet R : liste de (clef, valeur) en mémoire.

`mapreduce` applique à l'`input` les fonctions écrites en R : `map`, `combine` et `reduce` pour produire en `output`, un objet *big data*. Par défaut `map` est la fonction identité, `combine` et `reduce` sont `NULL`.

Les objets manipulés par les fonctions `to.dfs` et `from.dfs` transitant par la mémoire ne peuvent être très volumineux ; elles sont surtout utiles pour les tests.

2 Exemples élémentaires

2.1 Premiers tests

Chargement de la librairie `rmr2` et premières gestions d'objets big-data. Exécuter les instructions suivantes en prenant bien soin d'identifier à chaque fois les types d'objets manipulés. C'est sans doute la principale difficulté.

```
library(rmr2)
# pour utiliser RHadoop sans Hadoop
rmr.options(backend = "local")
# test de la librairie
from.dfs(to.dfs(1:10))
# liste de (clef, valeur)
keyval(1, 1:10)
# création d'un objet big data
objetBG=to.dfs(keyval(1, 1:10))
# retour à R
objetR=from.dfs(objetBG)
objetR
# extractions
keys(objetR)
values(objetR)
##########
# mtcars est un data frame contenant la
# variable nombre de cylindres,
# cette variable est définie comme clef
keyval(mtcars[,"cyl"], mtcars)
```

Deux exemples élémentaires utilisant la fonction `mapreduce`.

```
# carrés d'entiers
small.ints = to.dfs(1:10)
calc=mapreduce(
     input = small.ints,
# définition de la fonction map
     map = function(k, v) cbind(v, v^2))
# la fonction reduce est NULL par défaut
results=from.dfs(calc)
results
# même chose avec la somme des carrés
calc2=mapreduce(
     input = small.ints,
                  # définition de la fonction map
     map = function(k, v) keyval(1,v^2),
                  reduce = function(k,vv) sum(vv))
results=from.dfs(calc2)
values(results)
```

2.2 Comptage d'entiers

Il s'agit de compter les nombres d'occurrence de 50 valeurs suivant une loi de Bernoulli de paramètres 32 et 0,4. La fonction `tapply` le réalise en une ligne mais c'est encore une façon d'illustrer l'usage des couples (clef, valeur) de Mapreduce.

```
# échantillonnage binomial
# liste de 50 valeurs, sommes 32 bernouilli (p=0.4)
groups = rbinom(32, n = 50, prob = 0.4)
# effectif de chaque valeur
tapply(groups, groups, length)
```

Il s'agit ensuite de définir l'objet big data avant de définir les fonctions map et reduce.

```
groupe = to.dfs(groups)
result =  mapreduce(
    input = groupe,
# associer une paire (entier, 1) à chaque
# valeur ou entier présent dans ''groupe''
    map = function(., v) keyval(v, 1),
# par défaut, les clefs de même ''valeur'' sont
# associées dans un liste ou un vecteur.
# Pour une valeur de clef donnée, reduce compte
# le nombre de 1
    reduce = function(k, vv) keyval(k, length(vv)))
from.dfs(result)
```

2.3 Comptage de mots

Le nombre d'occurrences de mots dans un texte est l'usage canonique et l'exemple (*hello world*) le plus utilisé pour illustrer MapReduce. Le principe est le même que ci-dessus pour compter les entiers : construire des paires (clef, liste) où "clef" est un mot et "liste" une liste de 1 désignant chacun une occurrence d'un mot. L'étape map inclut un travail supplémentaire qui consiste à découpé le texte, ou simple chaîne de caractères, en mots. Un mot est simplement situé entre deux *patterns* ici des caractères "espace". L'étape `combine` est implicite avant l'exécution de la fonction `reduce`.

La fonction `mapreduce` est également incluse dans une fonction.

```
wordcount =
  function(input,pattern = " "){
          # définition de l'étape map
    wc.map = function(.,lines) {
    keyval(unlist(strsplit(
          x = lines,
          split = pattern)),1)}
    wc.reduce = function(word,counts ){
                # définition de l'étape reduce
          keyval(word, sum(counts))}
   mapreduce(input = input,
      map = wc.map,
      reduce = wc.reduce,
      combine = T)} # combine au niveau de reduce
```

Tester la fonction en entrant un texte rudimentaire.

```
texte0=c("ceci est un texte court vraiment
  court mais un texte quand même")
texte0
texte=to.dfs(texte0)
from.dfs(wordcount(texte))
```

3 Échantillonnage aléatoire simple

3.1 Objectif

Dans un environnement Hadoop de datamasse le premier objectif est de pouvoir disposer d'un programme d'échantillonnage aléatoire simple afin, par exemple, de pouvoir explorer les données avec des techniques et logiciels statistiques élémentaires puis construire un échantillon test et un échantillon d'apprentissage restreint pour exécuter en mémoire les programmes des méthodes bien connues d'apprentissage statistique. Cet objectif doit nécessairement être atteint en une seule passe, une seule lecture, de l'ensemble de la base.

3.2 Algorithme élémentaire

A titre pédagogique, voici un programme qui se contente de sélectionner les observations avec une probabilité p fixée *a priori*. Un échantillon est alors obtenu avec une taille approximative dépendant du nombre de processeurs pour l'étape map et de la connaissance plus ou moins précise de la taille totale. Cet algorithme mixte deux approches, une de *reservoir sampling* et une de tri d'un tirage de nombres aléatoires suivant une loi uniforme. Le principe général est donc de construire un "réservoir" de taille "raisonnable" en sélectionnant des observations en fonction du tirage d'une loi uniforme avant de trier ce réservoir puis pour finit de sélectionner les n première valeurs constituant l'échantillon final.

Simulation d'une (petite) base de données et extraction élémentaire d'un échantillon.

```
# Génération d'une base
set.seed(1)
donnee=data.frame(matrix(rnorm(5000),ncol=5))
# tirage aléatoire en mémoire
isamp=sample(1:1000,100,replace=FALSE)
donnee[isamp,]
# production de la datamasse
base=to.dfs(donnee)
```

Le contexte fonctionnel Mapreduce complique les choses.

```
aleaSimp=function(input,n,p){
# n: taille de l'échantillon voulu
# p: probabilité de conserver des observations
# dans le réservoir
## fonction de tri des lignes d'un data frame
arrange.matrix=function(x,...)
  as.matrix(arrange(as.data.frame(x),...))
mapreduce(
    input=input,
  # étapes map
    map =  function(k,v){
  # tirage selon loi uniforme
                  v=cbind(alea=runif(nrow(v)),v)
  # sélection des observations
              v=v[v[,"alea"]<p,]
  # tri
        v=arrange(v,alea)
  # émission
      keyval(1,v)},
  # étape reduce de tri et sélection
        # de n observations
        reduce=function(kk,vv){
  # tri et sélection
      vv=arrange(vv,alea)[1:n,]
  # émission
      vv=keyval(1,vv)},
        combine=TRUE)}
# exécution
from.dfs(aleaSimp(base,100,.2))
```

3.3 ScaSRS

La méthode (ScaSRS) (scalable simple random sampling) proposée par (Meng, 2013)[**?**] améliore l'algorithme précédent afin de mieux contrôler, par des bornes plus fines de p, tant la taille de l'échantillon, que celle du réservoir. Différentes stratégies sont proposées selon que la taille de la base est connue on non et selon que la paramètre fixé est le taux d'échantillonnage ou la taille de l'échantillon. Un paramètre à fixer (δ), définit le risque de ne pas obtenir un échantillon de taille au moins n. Si la valeur de δ est choisie trop grande, l'échantillon est bien de taille n mais l'algorithme produit un grand ensemble initial d'observations dont le tri est nécessairement long. En revanche, si δ est trop petit, l'échantillon ne sera pas complet.

Il est facile de modifier le programme précédent afin d'y insérer les différentes stratégies de ScaSRS. Cela revient à préciser la borne p de pré-sélection des observations. Il n'est pas non plus très difficile de compléter ces algorithmes afin de construire des versions stratifiées.

4 Régression

4.1 Principe

L'estimation des paramètres d'une régression en utilisant toute la base plutôt qu'un échantillon est l'occasion d'illustrer des opérations matricielles avec MapReduce.

Le programme est une simplification, de celui proposé dans le tuteuriel associé à RHadoop. Ainsi, pour calculer le produit $\mathbf{X}'\mathbf{X}$, cette "interface" R *vs.* Hadoop répartit automatiquement (à tester dans un vrai environnement Hadoop ?) sur les différents serveurs, les calculs des matrices $(p \times p)$: $\mathbf{x}_i'\mathbf{x}_i$ de l'instruction `map` et dont les résultats sont dans une liste avec un élément ou matrice par nœud. Chaque matrice contient l'ensemble des calculs pour les lignes de la matrice de *design* enregistrées dans le nœud en question. Les matrices de cette liste sont finalement sommées dans l'étape `reduce`.

4.2 Programme

Construction des données par simulation et calculs de la régression sans MapReduce.

```
# Génération des variables explicatives
X=matrix(rnorm(2000), ncol = 10)
# Génération de la variable à expliquer
y=as.matrix(rnorm(200))
# Data frame
base=data.frame("y"=y,X)
# Coefficients de la régression
lm(y~.,data=base)
# ou, c'est équivalent:
design=cbind(rep(1,200),X)
solve(t(design)%*%design,t(design)%*%y)
```

En supposant qu'une matrice de taille $p \times p$ tienne en mémoire ou encore que chaque nœud puisse calculer la somme partielle de $\mathbf{X}'\mathbf{X}$ des observations d'un mœud, le calcul de la régression avec MapReduce se fait en trois étapes :

1. calcul de $\mathbf{X}'\mathbf{X}$ où $\mathbf{X}$ est maintenant la matrice de design (avec la première colonne fixée à 1,
2. calcul de $\mathbf{X}'\mathbf{y}$,
3. résolution des équations normales $\mathbf{X}'\mathbf{X} = \mathbf{X}'\mathbf{y}$.

```
# construction de la ``Big base''
Bbase=to.dfs(base)
# Définition d'une fonction reduce
Sum = function(., XX)
    keyval(1, list(Reduce("+", XX)))
# Calcul de X'X par mapreduce
xtx=values(from.dfs(mapreduce(input=Bbase,
  map = function(., V) {
    Xdesign=cbind(rep(1,nrow(V)),as.matrix(V[,-1]))
    keyval(1, list(t(Xdesign) %*% Xdesign))},
  reduce = Sum,
  combine = TRUE)))[[1]]
# Calcul de X'y par mapreduce
xty=values(from.dfs(mapreduce(input=Bbase,
  map = function(., V) {
    y = as.matrix(V[,1])
    Xdesign=cbind(rep(1,nrow(V)),as.matrix(V[,-1]))
    keyval(1, list(t(Xdesign) %*% y))},
  reduce = Sum,
  combine = TRUE)))[[1]]
# résolution des équations normales
solve(xtx,xty)
```

Comparer avec les coefficients précédemment obtenus. Attention, cette stratégie ne donne que des résultats très partiels, sans statistique de test, ni diagnostic des résidus, ni possibilité de sélection de variables.

5 k-means

5.1 Principe

Comme pour la régression, voici une simplification du programme k-means du tuteuriel associé à RHadoop. Il s'agit cette fois d'itérer un nombre fixé a priori d'étapes MapReduce ou bien jusqu'à convergence de l'algorithme.

- La première étape `map` initialise l'algorithme en répartissant aléatoirement les observations en k classes puis gère l'affectation des observations aux centres les plus proches,
- l'étape `reduce` calcule les nouveaux centres.

5.2 Programme

Dans cette version simplifiée, le nombre d'itérations est fixé a priori et les numéros des classes finalement obtenus ne sont pas listés. Ceux-ci sont stockés comme clefs dans la base pour d'autres utilisations.

```
# fonction principale
kmeans.mr = function(P,num.clusters,num.iter){
# fonction à définir de calcul des distances
# ici euclidienne classique
dist.fun = function(C, P){
```

```
     apply(C,1,function(x)colSums((t(P) - x)^2))}
# fonction de l'étape map
km.map = function(., P){
    nearest = {
      if(is.null(C)) # initialisation le 1er tour
      sample(1:num.clusters,nrow(P),replace = TRUE)
      else {  # sinon cherche centre le plus proche
        D = dist.fun(C, P)
        nearest = max.col(-D)}}
        keyval(nearest, P)}
# fonction reduce de calcul des barycentres
# des observations associées à une même
# clef (classe)
km.reduce = function(., G) t(as.matrix(apply(G, 2,
            mean)))
# programme principal
## initialisation
C = NULL
##itérations
    for(i in 1:num.iter ) {
      res = from.dfs(
          mapreduce(P,
            map = km.map,
            reduce = km.reduce))
            C=values(res)
   ###  si des centres ont disparu
      if(nrow(C) < num.clusters) {
        C = rbind(C,matrix(rnorm(
         (num.clusters-nrow(C))*nrow(C)),
          ncol = nrow(C))%*%C)}}
        C}
```

Tester sur des données simulées.

```
# simulation de 5 centres + bruit gaussien
set.seed(1)
P=do.call(rbind,rep(list(matrix(rnorm(10,
   sd = 10),ncol=2)),20))+
   matrix(rnorm(200), ncol =2)
# test
out = kmeans.mr(to.dfs(P),
        num.clusters = 5,
        num.iter = 8)
plot(P)
points(out,col="blue",pch=16)
```