TP RHadoop

J. Bigot / A. Richou

23/09/2016

1 Introduction

1.1 Présentation du TP

L'objectif de ce TP est de se familiariser au paradigme MapReduce. Il existe à l'heure actuelle de nombreuses technologie concurrentes pour traiter de gros jeux de données mais pour éviter les coûts d'entrée importants en terme de connaissances nous nous restreignons à l'utilisation de la librairie `rmr2` de RHadoop. Cette dernière possède l'avantage d'être simple d'accès car elle repose sur le langage R et ne nécessite pas l'installation d'un serveur Hadoop. Ce TP s'inspire très largement du TP disponible sur le site WikiStat[1] qui s'inspire lui même très largement du tuteuriel proposé par RHadoop[2].

1.2 Installation

La librairie `rmr2` n'est pas disponible sur le CRAN, il faut donc réaliser l'installation à la main. Téléchargez la dernière version de `rmr2` sur l'espace collaboratif *GitHub* et l'installer. Par exemple pour Linux il suffit d'aller dans le dossier built et de taper la commande

```
R CMD INSTALL rmr2_3.3.0.tar.gz
```

Pour windows il faut utiliser le paquet `rmr2_3.3.0.zip`. Le chargement de `rmr2` dépend d'autres librairies dont l'installation ne pose pas de problème à l'exception de `rjava` qui peut s'avérer compliquée sous Windows.

1.3 Fonctions

Comme d'habitude il conviendra d'utiliser l'aide en ligne pour avoir une description détaillée des diiférentes fonctions. Tout d'abord les données stockées au format HDFS sont du type `big.data.object`. Voici quelques fonctions qui seront utiles pour la suite.

- 'keyval' crée des paires (clé,valeur). 'values' et 'keys' permet d'extraire les clés et les valeurs de ces paires.
- 'to.dfs(kv)' où 'kv' est une liste de (clé,valeur) ou un objet R (dans ce cas la clé est 'NULL') permet de transformer 'kv' en un objet 'big.data.object'. Par défaut cet objet est déposé dans un fichier temporaire mais il est possible de spécifier le chemin d'un fichier existant.
- 'from.dfs(input)' permet de transformer l'object big data 'input' en un objet R de type liste de (clé,valeur) en mémoire.
- la fonction 'mapreduce' permet d'appliquer un traitement MapReduce à des données. Son fonctionnement précis sera détaillé par la suite. Par défaut 'map' est la fonction identité , 'combine' et 'reduce' sont 'NULL'.

[1] http://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-tutor5-R-mapreduce.pdf
[2] https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md

2 Exemples élémentaires

2.1 *Ab Initio*

On commence par charger la librarie `rmr2` et spécifier l'utilisation en locale, i.e. sans serveur Hadoop.

```
library(rmr2)
rmr.options(backend = "local")
```

Exécuter les instructions suivantes en prenant soin de bien identifier les types d'objets manipulés.

```
# création d'un objet de type big data
test <- to.dfs(1:10)
# retour à R
test2 <- from.dfs(test)
# création d'une liste de (clef,valeur)
test3 <- keyval(1,1:10)
keys(test3)
values(test3)
# mtcars est un data frame contenant la variable
# nombre de cylindres. Cette variables est définie comme
# clé, la valeur associée est la ligne correspondante du data frame
keyval(mtcars[,"cyl"],mtcars)
```

Il faut bien comprendre que la fonction `to.dfs` n'est utile que pour une utilisation locale de `rmr2`. Pour une utilisation réelle, on utilisera des fichier hdfs préexistants.

On donne maintenant deux exemples d'utilisation de la fonction `mapreduce`. Le premier exemple consiste à calculer des carrées d'entiers.

```
# carrés d'entiers
entiers <- to.dfs(1:10)
calcul.map = function(k,v){
  cbind(v,v^2)
}
calcul <- mapreduce(
  input = entiers,
  map = calcul.map
  # la fonction reduce est nulle par defaut
  )
resultat <- from.dfs(calcul)
resultat
```

Le deuxième exemple consiste à calculer la somme de carrés d'entiers.

```
calcul2.map = function(k,v){
  keyval(1,v^2)
}
calcul2.reduce = function(k,v){
  sum(v)
}
calcul2 <- mapreduce(
  input = entiers,
```

```
  map = calcul2.map,
  reduce = calcul2.reduce)
resultat2 <- from.dfs(calcul2)
resultat2
```

Un dernier exemple pour la route.

```
entiers3 <- to.dfs(1:1000000)
calcul3.map = function(k,v){
  keyval(1,1)
}
calcul3.reduce = function(k,v){
  sum(v)
}
calcul3 <- mapreduce(
  input = entiers3,
  map = calcul3.map,
  reduce = calcul3.reduce
  )
resultat3 <- from.dfs(calcul3)
resultat3
```

Question 1 *Comment expliquez-vous la valeur de* `resultat3` *?*

2.2 Comptage d'entiers

Il s'agit de compter les nombres d'occurence de 50 tirages d'une loi de Bernoulli de paramètres 32 et 0,4. La fonction `tapply` le réalise en une seule ligne mais c'est encore un exemple didactique illustrant l'utilisation du paradigme mapreduce.

```
tirage <- to.dfs(rbinom(32,n=50,prob=0.4))
# le map associe à chaque entier une paire (entier,1)
comptage.map = function(k,v){
  keyval(v,1)
}
comptage.reduce = function(k,v){
  keyval(k,length(v))
}
comptage <- mapreduce(
  input = tirage,
  map = comptage.map,
  reduce = comptage.reduce)
from.dfs(comptage)
table(values(from.dfs(tirage)))
```

2.3 *Salve, munde!*

Il est temps maintenant de présenter l'exemple canonique (*hello world*) de MapReduce qui consiste à compter les mots d'un texte. Le principe est le même que pour le comptage d'entiers, à la différence près que l'étape map requiert plus de travail préparatoire pour découper le texte en mots.

```
#On définit une fonction wordcount pour le comptage de mots
wordcount = function(input,pattern = " "){
  #input : texte à analyser au format big data
  #pattern : sigle utilisé pour la séparation des mots
  #          (" " par défaut)
  wordcount.map = function(k,texte){
    keyval(unlist(strsplit(x = texte, split = pattern)),1)
  }
  wordcount.reduce = function(word,count){
    keyval(word, sum(count))
  }
  resultat<-mapreduce(
    input = input,
    map = wordcount.map,
    reduce = wordcount.reduce)
  return(resultat)
}

#Un exemple d'utilisation avec un texte simple
texte = c("un petit texte pour l'homme mais un grand grand grand texte pour l'humanité")
from.dfs(wordcount(to.dfs(texte)))
```

3 k-means

3.1 Principe

On rappelle succintement le principe du k-means.

- On a n points de $\mathbb{R}^d$ pour lesquels on cherche k classes en faisant r itérations.
- On initialise en affectant à chaque point une classe aléatoirement (uniformément sur $\{1, ..., k\}$).
- On calcule les barycentres de chaque classe.
- On reaffecte une classe à chaque point en déterminant le barycentre le plus proche puis on recalcule les barycentres de chaque classe. On itère $r - 1$ fois cette étape.

Dans le cadre MapReduce, on va itérer r fois une fonction `mapreduce`.

- La première étape 'map' consiste à initialiser les affectations de classes. Les étapes 'map' suivantes gèrent les affectations de classes.
- Les étapes 'reduce' consistent à calculer les barycentres de chaque classes.

3.2 Programme

```
#fonction principale
kmeans = function(Pts,nbClusters,nbIter){
  #P : pts (au format big data) sur lesquels on fait un k-mean
  # nbClusters : nombre de clusters désirés
  # nbIter : nombre d'itérations dans le k-mean
```

```
  # retourne C : matrice des pts du k-mean

  #calcul de distance
  distance = function(c,P){
    #determine pour chaque point de P
    #la distance a chaque point de c
    resultat <- matrix(rep(0,nrow(P)*nrow(C)), nrow(P), nrow(C))
    for (i in 1:nrow(c)){
      resultat[,i] <- rowSums(
        (P-matrix(rep(c[i,],nrow(P)),nrow(P), ncol(P), byrow = TRUE))^2)
    }
    return(resultat)
  }
  #fonctions map : initialisation ou détermination des
  # centres les plus proches
  km.map = function(.,P){
    if (is.null(C)){
      #initialisation au premier tour
      pproche <- sample(1:nbClusters,nrow(P),replace = TRUE)
    }
    else {
      #sinon on cherche le plus proche
      D <- distance(C,P)
      pproche <- max.col(-D)
    }
    keyval(pproche,P)
  }
  #fonction reduce : calcul des barycentres
  km.reduce = function(.,G){
    t(as.matrix( apply(G,2,mean) ))
  }
  #programme principal
  ## initialisation
  C <- NULL
  ## iterations
  for(i in 1:nbIter){
    resultat = from.dfs(mapreduce(
      input = Pts,
      map = km.map,
      reduce = km.reduce))
    C <- values(resultat)
    ## si des centres ont disparu
    ## on en remet aléatoirement
    if (nrow(C) < nbClusters){
      C = rbind(C,matrix( rnorm((nbClusters-nrow(C))*nrow(C)),
                          ncol = nrow(C))%*%C)
    }
  }
  return(C)
}
```

On peut tester le résultat sur des données simulées.

```
#simulation de 5 centres plus bruit gaussien
#on reinitialise le generateur pour faciliter le debogage
set.seed(1)
P <- matrix(rep(0,200),100,2)
for (i in 1:5){
  P[((i-1)*20+1):(i*20),] <- (matrix(rep(rnorm(2, sd = 20),20),ncol=2,byrow = TRUE) +
                                matrix(rnorm(40),ncol=2))
}
#test
resultat <- kmeans(to.dfs(P),5,8)
plot(P)
points(resultat,col="blue",pch=16)
```

Question 2 *En pratique l'algorithme* `kmeans` *donné précédemment ne peut pas fonctionner correctement sur de gros volumes de données et nécessite l'ajout d'une étape "combine". Expliquez pourquoi et modifiez le code afin d'incorporer cette nouvelle étape.*

4 Régression linéaire

4.1 Principe

On considère un modèle linéaire du type $y = xA + \varepsilon$ avec $\varepsilon unbruit$,y$ de taille $p \times 1$, x de taille $p \times k$, A de taille $p \times 1$ et on suppose que k est "petit" et p très grand. L'estimateur des moindre carrés de A est donné par $(x'x)^{-1}x'y$. On va utiliser le principe map reduce pour calculer dans un premier temps $x'x$ et $x'y$ qui ont le bon gout d'avoir des tailles raisonnables puis on termine le calcul sous R pour calculer $(x'x)^{-1}x'y$.

4.2 Code

```
#fonction principale
regression = function(data){

  #fonction map pour XtX
  xtx.map = function(.,v){
    X <- as.matrix(cbind( matrix(rep(1,nrow(v)),ncol=1) , v[,-1] ))
    keyval(1, list(t(X)%*%X))
  }
  #fonction reduce pour XtX
  xtx.reduce = function(.,v){
    keyval(1,Reduce("+", v))
    #keyval(1,v)
  }
  #Calcul de XtX
  xtx = mapreduce(
    input = data,
    map = xtx.map,
    reduce = xtx.reduce)
  #fonction map pout Xty
  xty.map = function(.,v){
    y <- v[,1]
    X <- cbind(matrix(rep(1,nrow(v)),ncol=1) , v[,-1] )
```

```
    keyval(1, list(t(X)%*%y))
  }
  #Calcul de Xty
  xty = mapreduce(
    input = data,
    map = xty.map,
    reduce = xtx.reduce)
  #Regression linéaire
  xtx <- values(from.dfs(xtx))
  xty <- values(from.dfs(xty))
  return(solve(xtx,xty))
}
```

Essai sur un exemple.

```
#on reinitialise le generateur pour faciliter le debogage
set.seed(1)
#generation des variables explicatives
X <- matrix(rnorm(3000), ncol=10)
#generation des variables à expliquer
y <- X%*%( as.matrix(rep(c(1,2),5)) ) + matrix(rnorm(300),ncol=1)
# data frame
base <- data.frame("y"=y,X)
resultat1 <- regression(to.dfs(base))
resultat1
resultat2 <- lm(y~.,data=base)
resultat2
```

Une remarque pour terminer : si l'exemple n'est pas suffisament gros (200 lignes au lieu de 300 lignes par exemple) alors il n'y aura qu'une étape map et qu'une étape reduce ce qui peut créer des problèmes de debogage.