1 Objectif

Technologie MapReduce sous R avec le package « rmr2 ».

« Big Data[1] » (« mégadonnées » ou « données massives » en français), en veux-tu en voilà. Tout le monde en parle, c'est le sujet à la mode. Il suffit de voir l'évolution des requêtes associées sur Google Trends pour s'en rendre compte[2]. Leur valorisation est un enjeu fort, on parle de « big data analytics »[3]. Dans les faits, il s'agit d'étendre le champ d'application des techniques de statistique exploratoire et de data mining à de nouvelles sources de données dont les principales caractéristiques sont la volumétrie, la variété et la vélocité.

Un esprit chagrin serait tenté de dire qu'il n'y rien de bien nouveau là-dedans. La volumétrie faisait déjà partie intégrante de la problématique data mining, il en est de même pour la variété avec le « text mining » par exemple, ou encore la vélocité avec le « data stream mining ». C'est vrai et faux à la fois. On avait vu fleurir le même type de commentaires de la part des statisticiens lors de l'émergence du data mining dans le milieu des années 90. La réponse qui m'a le plus convaincue est celle de Gregory Piatetsky-Shapiro sur le site KDnuggets. Il explique en substance que la nouveauté réside dans le fait que les caractéristiques additionnelles des données deviennent une partie du problème à résoudre[4]. Dans mon cours d'Introduction au Data Science (un autre terme « fashion » qu'on a parfois du mal à cerner[5]), j'ai moi-même essayé de situer l'évolution « statistique exploratoire - data mining - big data analytics » sous l'angle de l'évolution des sources d'informations que l'on appréhende. Sources qui, elles-mêmes, évoluent en fonction de l'évolution de la technologie, des outils et pratiques de stockage des données[6].

L'informatique distribuée est un pilier essentiel du big data. Il est illusoire de vouloir augmenter à l'infini la puissance des serveurs pour suivre la croissance exponentielle des informations à

[1] http://en.wikipedia.org/wiki/Big_data

[2] http://www.google.fr/trends/explore#q=big%20data - Depuis 2012, l'évolution est quasi-exponentielle.

[3] http://www.sas.com/en_us/insights/analytics/big-data-analytics.html

[4] KDnuggets Polls, « Has Big Data significantly changed Data Science principles and practice ? », octobre 2013.

[5] http://en.wikipedia.org/wiki/Data_science

[6] Tutoriel Tanagra, « Introduction au Data Science - Du data mining au big data - Enjeux et opportunités », mai 2014. Je pose notamment la question des nouveaux enseignements que l'on pourrait introduire dans nos maquettes de M1 / M2.

traiter. La solution passe par une coopération efficace d'une myriade de machines connectées en réseau, assurant à la fois la gestion de la volumétrie et une puissance de calcul décuplée. « Hadoop » s'inscrit dans ce contexte. Il s'agit d'un framework Java libre de la fondation Apache destiné à faciliter la création d'applications distribuées et échelonnables[7]. « Distribuées » signifie que le stockage et les calculs sont réalisés à distance sur un cluster (groupe) de nœuds (de machines). « Echelonnables » dans le sens où si l'on a besoin d'une puissance supplémentaire, il suffit d'augmenter le nombre de nœuds sans avoir à remettre en cause les programmes et les architectures. Dans cette optique, MapReduce[8] de Hadoop joue un rôle important. C'est un modèle de programmation qui permet de distribuer les opérations sur des nœuds d'un cluster. L'idée maîtresse est la subdivision des tâches que l'on peut dispatcher sur des machines distantes, ouvrant ainsi la porte au traitement de très grosses volumétries. Une série de dispositifs sont mis en place pour assurer la fiabilité du système (ex. en cas de panne d'un des nœuds).

Dans ce tutoriel, nous nous intéressons à la programmation MapReduce sous R. Nous nous appuierons sur la technologie RHadoop[9] de la société Revolution Analytics. Le package « rmr2 » en particulier permet de s'initier à la programmation MapReduce sans avoir à installer tout l'environnement, tâche qui, en soi, est déjà suffisamment compliquée comme cela. Il existe des tutoriels consacrés à ce thème sur le net. Celui Hugh Devlin (janvier 2014) en est une illustration[10]. Mais il s'adresse à un public connaisseur des statistiques et de la programmation R, en commençant notamment par faire le parallèle (c'est de circonstance) avec la fonction lapply() que les apprentis du langage R ont souvent bien du mal à appréhender. Au final, il m'a fallu beaucoup de temps pour réellement saisir la teneur des fonctions étudiées. J'ai donc décidé de reprendre les choses à zéro en commençant par des exemples très simples dans un premier temps, avant de progresser jusqu'à la programmation d'algorithmes de data mining, la régression linéaire multiple en l'occurrence.

[7] http://fr.wikipedia.org/wiki/Hadoop

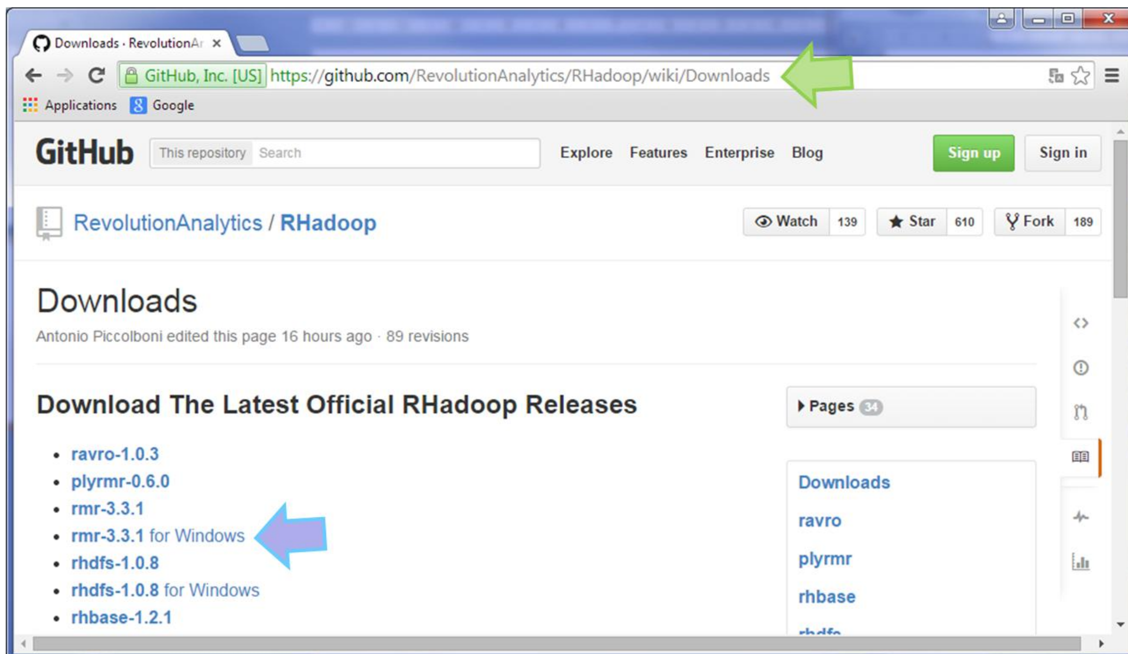
[8] http://fr.wikipedia.org/wiki/MapReduce

[9] http://blog.revolutionanalytics.com/2011/09/mapreduce-hadoop-r.html

[10] https://github.com/RevolutionAnalytics/rmr2/blob/master/docs/tutorial.md

2 Installation du package « rmr2 »

Un tutoriel décrit la procédure pour Windows[11]. Il semble qu'il faut être en 64 bits pour espérer faire fonctionner le dispositif. Pour ma part, j'ai chargé le package « functional », puis j'ai récupéré la librairie « rmr2 » sur GitHub (**rmr2_3.3.1.zip**). Je l'ai installée manuellement à l'aide de la commande « Installer des packages depuis des fichiers zip ».



A titre d'information, je suis sous Windows 7 – 64 bits et j'ai réalisé toutes mes expérimentations sous **R 3.1.2**.

Nous devons alors exécuter les commandes suivantes pour préparer le terrain.

```
#charger la librairie rmr2
library(rmr2)

#fonctionner sans l'environnement hadoop
rmr.options(backend="local")
```

La seconde commande nous donne l'opportunité de s'exercer à la programmation MapReduce sans avoir à installer l'environnement Hadoop.

[11] http://tuxette.nathalievilla.org/?p=1455&lang=en#win

3 Le concept MapReduce – Transmission d'un vecteur

Dixit Wikipédia, le principe MapReduce consiste en deux fonctions (deux phases) map() et reduce(). Dans l'étape **MAP**, le nœud analyse le problème, le découpe en sous-problèmes, et les délègue à d'autres nœuds (qui peuvent aussi en faire de même récursivement). A chaque sous-problème est associée une clé qui permet de le repérer. Les sous-problèmes sont traités à l'aide de la fonction **REDUCE**. Elle renvoie les résultats en leur associant également une clé. Le couple < clé, valeur > tient donc une place extrêmement importante dans le dispositif MapReduce, « valeur » étant relatif - selon le cas - à des données ou des résultats de calculs. Nous y accorderons une attention particulière dans les exemples de ce tutoriel.

3.1 Premier traitement – Comptage des valeurs

Dans cette section, nous souhaitons manipuler un vecteur d'entiers. Nous souhaitons distinguer les valeurs paires et impaires, les envoyer sur 2 nœuds différents, et effectuer un traitement sur chacun des sous ensembles. Voici le vecteur à traiter.

```
#vecteur de valeurs entières
x <- c(2,6,67,85,7,9,4,21,78,45)
```

Nous observons 10 entiers, avec respectivement 4 et 6 valeurs paires et impaires.

MAP. Nous écrivons la fonction **map**() suivante. Pour une valeur donnée, elle teste si elle est paire en analysant le reste de la division par 2. Si le reste est égal à 0 c.-à-d. la valeur est paire, la fonction lui attribue la clé 1, sinon elle lui attribue la clé 2.

```
#map selon la parité de la valeur
map_valeurs <- function(., v){
  #calcul de la clé
  cle <- ifelse (v %% 2 == 0, 1, 2)
  #retour des vecteurs clé et valeurs
  return(keyval(cle,v))
}
```

La fonction prend usuellement deux entrées : la clé et la valeur à traiter. Dans notre cas, nous nous servons de la valeur pour générer la clé. Le premier paramètre est donc ignoré.

Autre information très importante, « **v** » dans notre cas représente le vecteur « **x** » à analyser. La variable « **cle** » générée par **ifelse()** est donc un vecteur.

Le rôle de **keyval()** qui est renvoyée en sortie par **return()** est fondamental. Il associe une **clé** à chaque **valeur** en entrée de la fonction. A l'issue du traitement, « v » étant un vecteur, nous renvoyons 2 vecteurs dans **keyval()**, « cle » et « v » :

cle	v
1	2
1	6
2	67
2	85
2	7
2	9
1	4
2	21
1	78
2	45

MAPREDUCE se base sur ces 2 vecteurs pour dispatcher les données en 2 sous-vecteurs : (2, 6, 4, 78) pour les valeurs paires correspondant à la clé = 1 ; (67, 85, 7, 9, 21, 45) pour les impaires avec la clé = 2.

REDUCE. Dans la phase reduce, les sous-ensembles sont traités sur les nœuds du cluster. La fonction **reduce**() est donc appelée autant de fois qu'il y a de valeurs différentes de clés.

```
#reduce sur chaque sous-vecteur
reduce_valeurs <- function(k, v){
  #longueur du vecteur à traiter
  nb <- length(v)
  #renvoyer la clé et le résultat associé
  return(keyval(k, nb))
}
```

La clé est nécessaire pour savoir quel sous-vecteur on traite. Nous calculons sa longueur avec **length()**. Nous renvoyons le résultat avec **return()** en associant la clé et le résultat des calculs avec la fonction **keyval()**.

MAPREDUCE avec rmr2. Voyons maintenant comment organiser tout cela à l'aide de la fonction **mapreduce()** de rmr2.

```
#transformation en type compatible rmr2
x.dfs <- to.dfs(x)

#fonction mapreduce de rmr2
calcul <- mapreduce(input = x.dfs, map = map_valeurs, reduce = reduce_valeurs)

#transformation en un type R usuel
resultat <- from.dfs(calcul)

#affichage
print(resultat)

#classe de resultat
print(class(resultat))
```

to.dfs() sert à passer les données du format R en un type reconnu par rmr2 ; **from.dfs()** réalise la transformation inverse.

La fonction **mapreduce()** est le cœur du dispositif. Dans notre cas, il prend 3 paramètres :

- « input » correspond aux données à traiter ;
- « map » est la fonction appelée pour mapper les données en clé - valeurs ;
- « reduce » se charge de réaliser les calculs sur les portions de données.

Nous obtenons à l'issue des traitements sous R :

```
> #affichage
> print(resultat)
$key
[1] 1 2

$val
[1] 4 6

>
> #classe de resultat
> print(class(resultat))
[1] "list"
```

« resultat » est de type « list ». Il contient: le vecteur des clés (nommé « **key** ») et le vecteur des valeurs calculées (nommé « **val** »). Il y a 4 valeurs paires (clé 1) dans le vecteur initial « x », et 6 valeurs impaires (clé 2).

Nous pouvons accéder spécifiquement aux clés et aux valeurs calculées en utilisant les fonctions **keys()** et **values()** :

```
#affichage des clés
print(keys(resultat))

#affichage des valeurs calculées
print(values(resultat))
```

Nous avons :

```
> #affichage des clés
> print(keys(resultat))
[1] 1 2
>
> #affichage des valeurs calculées
> print(values(resultat))
[1] 4 6
```

3.2 Tracer l'exécution de MAPREDUCE

Pour tracer le déroulement des opérations, nous avons insérer des **print()** de contrôle dans les fonctions map() et reduce(). A savoir

```
#map selon la parité de la valeur
map_valeurs <- function(., v){
  #affichage de contrôle
  print("map") ; print(v)
  #calcul de la clé
  cle <- ifelse (v %% 2 == 0, 1, 2)
  #retour des vecteurs clé et valeurs
  return(keyval(cle,v))
}


#reduce sur chaque sous-vecteur
reduce_valeurs <- function(k, v){
  #affichage de contrôle
  print("reduce") ; print(k) ; print(v)
  #longueur du vecteur à traiter
  nb <- length(v)
  #renvoyer la clé et le résultat associé
  return(keyval(k, nb))
}
```

Lors de l'exécution du programme, nous voyons affiché dans la console :

```
[1] "map"
 [1]  2  6 67 85  7  9  4 21 78 45
[1] "reduce"
[1] 1
[1]  2  6  4 78
[1] "reduce"
[1] 2
[1] 67 85  7  9 21 45
```

La fonction map() est appelée une seule fois. Le vecteur complet lui est transmis. La fonction reduce() est appelée 2 fois : 1 appel pour chaque item de la clé, elle est accompagnée du sous-vecteur de données à traiter.

4 Transmission d'un data frame

Nous souhaitons calculer la somme des carrés des résidus d'une ANOVA (analyse de variance) dans cette section. L'originalité réside dans la manipulation et la transmission d'un data frame vers les nœuds. Le schéma ci-dessus est tout à fait transposable à cette nouvelle configuration. C'est le data frame maintenant qui sera tronçonné en plusieurs portions.

Préparation des données. Nous construisons notre jeu de données comme suit :

```
#vecteur indicateur de groupe
y <- factor(c(1,1,2,1,2,3,1,2,3,2,1,1,2,3,3))
#vecteur de valeurs
x <- c(0.2,0.65,0.8,0.7,0.85,0.78,1.6,0.7,1.2,1.1,0.4,0.7,0.6,1.7,0.15)
#construire un data frame
don <- data.frame(cbind(y,x))
```

Voici donc notre tableau de données (format interne **data.frame**) :

y	1	1	2	1	2	3	1	2	3	2	1	1	2	3	3
x	0.2	0.65	0.8	0.7	0.85	0.78	1.6	0.7	1.2	1.1	0.4	0.7	0.6	1.7	0.15

De nouveau, nous devons spécifier les fonctions map() et reduce().

MAP. La variable Y est l'indicateur de groupe, elle sert directement à définir les clés.

```
#map - dispatcher selon y
map_ssq <- function(., v){
  #la colonne y est la clé
  cle <- v$y
  #renvoyer la clé et le data frame entier
  return(keyval(cle,v))
}
```

C'est bien le data frame que vous renvoyons accompagnée de la clé en sortie de la fonction avec **keyval**().

REDUCE. Le data frame initial est découpé en plusieurs sous-parties correspond aux sous-ensembles d'observations définis par la variable Y. Nous calculons la somme des carrés des écarts à la moyenne sur le champ « x » du data frame « v » passé en paramètre.

```
#reduce - calcul
reduce_ssq <- function(k,v){
  #compter le nombre de ligne du data frame
  n <- nrow(v)
  #calcul sur le champ (colonne) x du data.frame
  ssq <- (n-1) * var(v$x)
  #renvoyer la clé et le résultat
  return(keyval(k,ssq))
}
```

C'est bien un data frame qui vient en entrée, nous utilisons **nrow()** et non length() pour accéder au nombre de lignes. Nous utilisons l'opérateur « $ » pour accéder au champ « x ».

Nous renvoyons en sortie la clé et le résultat du calcul.

Calculs. Il ne reste plus qu'à lancer les calculs.

```
#format rmr2
don.dfs <- to.dfs(don)

#mapreduce
calcul <- mapreduce(input=don.dfs,map=map_ssq,reduce=reduce_ssq)

#récupération
resultat <- from.dfs(calcul)
print(resultat)
```

Nous obtenons les sommes des carrés des écarts à l'intérieur de chaque sous-population.

```
$key
[1] 1 2 3

$val
[1] 1.152083 0.142000 1.293675
```

Nous effectuons la somme pour obtenir la somme des carrées des résidus.

```
#SSR
ssr <- sum(resultat$val)
print(ssr)
```

Nous obtenons la valeur **SSR = 2.587758**.

C'est le bon résultat que l'on peut obtenir avec la fonction AOV de R par exemple.

```
> #contrôle
> print(aov(x ~ y))
Call:
   aov(formula = x ~ y)

Terms:
                       y Residuals
Sum of Squares  0.149015  2.587758
Deg. of Freedom        2        12

Residual standard error: 0.4643776
Estimated effects may be unbalanced
```

Impression de contrôle. Encore une fois, lorsque nous introduisons une impression de contrôle dans la fonction reduce()…

```
#reduce - calcul
reduce_ssq <- function(k,v){
  #impression de contrôle
  print("reduce") ; print(v)
  #compter le nombre de ligne du data frame
  n <- nrow(v)
  #calcul sur le champ (colonne) x du data.frame
  ssq <- (n-1) * var(v$x)
  #renvoyer la clé et le résultat
  return(keyval(k,ssq))
}
```

Nous constatons qu'elle est appelée 3 fois pour les 3 portions de data frame.

```
[1] "reduce"
   y    x
1  1 0.20
2  1 0.65
4  1 0.70
7  1 1.60
11 1 0.40
12 1 0.70
[1] "reduce"
   y    x
3  2 0.80
5  2 0.85
8  2 0.70
10 2 1.10
13 2 0.60
[1] "reduce"
   y    x
6  3 0.78
9  3 1.20
14 3 1.70
15 3 0.15
```

reduce(1)

reduce(2)

reduce(3)

5 Algorithme de régression linéaire

J'ai mis du temps avant de comprendre la régression linéaire à la sauce Hugh Develin. En fait, il n'utilise qu'une seule valeur de clé. Ca paraît bien étrange dans notre schéma MapReduce sous R. La fonction reduce() ne sera appelée qu'une seule fois. Je ne vois pas très bien à quel stade apparaît la subdivision des calculs… je ne suis pas allé plus loin, l'idée reste à creuser[12].

Pour l'heure, je propose une autre piste dans cette section. Les données sont découpées en 2 blocs par la fonction map() (avec 2 valeurs de clés distinctes – la généralisation à K blocs ne pose pas de problèmes, il suffit de modifier la fonction map). Les calculs sont réalisés pour chaque bloc par la fonction reduce(). A la sortie, nous effectuons la consolidation en additionnant les matrices. Nous profiterons également de cet exemple pour aller plus loin dans la manipulation des données. Plutôt que de renvoyer une valeur atomique à la sortie de la fonction reduce(), nous enverrons une structure un peu plus complexe. Nous pourrons ainsi évaluer la souplesse de l'outil lorsqu'il s'agit d'aller vers des traitements plus élaborés.

Données. Nous utilisons les données **mtcars** [`data(mtcars)`]. Nous cherchons à expliquer la consommation (**mpg**) en fonction des autres variables.

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
Cadillac Fleetwood	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
Lincoln Continental	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
Chrysler Imperial	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
Fiat 128	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
Toyota Corolla	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
Toyota Corona	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
Dodge Challenger	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
AMC Javelin	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
Camaro Z28	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
Pontiac Firebird	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
Fiat X1-9	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
Porsche 914-2	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
Lotus Europa	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
Ford Pantera L	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
Ferrari Dino	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
Maserati Bora	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
Volvo 142E	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

[12] En fait, son idée serait intéressante si nous avions plusieurs sources de données ou si les données sont passées en blocs (il le dit mais il ne le montre nulle part). A chaque source correspond un appel à la fonction map() qui réalise les calculs sur cette portion de données, puis envoie le résultat dans une liste avec la même valeur de clé pour annoncer que ces matrices intermédiaires sont relatives au même problème. La fonction reduce() se charge alors d'effectuer les additions matricielles…

MAP. La stratégie map() consiste à subdiviser les données – le data frame – en plusieurs parties. Voici le code pour une partition aléatoire en 2 portions à peu près égales.

```
#map
map_lm <- function(., D){
  #génération de valeurs aléatoires
  alea <- runif(nrow(D))
  #clé - découpage en 2 parts à peu près égales
  #on peut facilement multiplier les sous-groupes
  cle <- ifelse(alea < 0.5, 1, 2)
  #renvoyer la clé et les données
  return(keyval(cle,D))
}
```

Remarque 1 : Le caractère aléatoire de la partition n'est pas obligatoire dans le contexte de la régression. Nous aurions pu tout aussi bien prendre les n_1 premiers individus pour la 1ère portion et les n_2 suivants pour la seconde (avec taille d'échantillon = n = $n_1 + n_2$). Par conséquent, si les fragments de données sont situés sur des machines différentes, il sera tout à fait possible d'effectuer les calculs localement avant de consolider les résultats.

Remarque 2 : La généralisation en une subdivision en K sous-groupes d'observations ne pose absolument aucun problème. Ainsi, le code reduce() et la consolidation qui suivent fonctionneront quel que soit le nombre de nœuds sollicités.

REDUCE. Penchons-nous un peu sur l'estimation des moindres carrés ordinaires (MCO) avant de décrire la fonction reduce(). Le modèle s'écrit :

$$y = Xa + \varepsilon$$

Y est la variable cible ; X est la matrice correspondant aux variables prédictives, une première colonne de valeurs 1 est accolée à la matrice pour tenir compte de la constante de la régression ; a est le vecteur des paramètres ; ε est le terme d'erreur qui résume les insuffisances du modèle.

L'estimateur des moindres carrés ordinaires â est défini par la formule :

$$\hat{a} = (X^t X)^{-1} X^t y$$

Où X^t est la transposée de la matrice X.

Regardons de près les coefficients des matrices pour comprendre la décomposition des calculs. Pour ($X^t X$), au croisement des variables X_j et X_m, nous avons :

$$\sum_{i=1}^{n} x_{ij} \times x_{im}$$

Les termes étant additifs, nous pouvons fractionner les calculs en 2 parties :

$$\sum_{i=1}^{n_1} x_{ij} \times x_{im} + \sum_{i=n_1+1}^{n} x_{ij} \times x_{im}$$

Il en est de même pour $(X^t y)$, au croisement de X_j et y :

$$\sum_{i=1}^{n} x_{ij} \times y_i = \sum_{i=1}^{n_1} x_{ij} \times y_i + \sum_{i=n_1+1}^{n} x_{ij} \times y_i$$

Subdiviser les calculs en K parties ne pose absolument aucun problème au regard de ces propriétés. Nous les exploitons (ces propriétés) pour écrire la fonction reduce() :

```
#reduce
reduce_lm <- function(k,D){
  #nombre de lignes
  n <- nrow(D)
  #récupération de la cible
  y <- D$mpg
  #prédictives
  X <- as.matrix(D[,-1])
  #rajouter la constante en première colonne
  X <- cbind(rep(1,n),X)
  #calcul de X'X
  XtX <- t(X) %*% X
  #calcul de X'y
  Xty <- t(X) %*% y
  #former une structure de liste
  res <- list(XtX = XtX, Xty = Xty)
  #renvoyer le tout
  return(keyval(k,res))
}
```

La nouvelle subtilité est que nous utilisons une liste pour renvoyer les deux matrices $(X^t X)$ et $(X^t y)$. Il faudra être très attentif lorsqu'il faudra consolider les résultats pour former les matrices globales correspondantes.

Calculs et récupération des résultats. Il ne reste plus qu'à la lancer les calculs…

```
#format rmr2
don.dfs <- to.dfs(mtcars)
#mapreduce
```

```
calcul <- mapreduce(input=don.dfs,map=map_lm,reduce=reduce_lm)
#récupération
resultat <- from.dfs(calcul)
print(resultat)
```

Voyons en détail l'objet « résultat » :

```
> print(resultat)
$key
[1] 1 1 2 2

$val
$val$XtX
              cyl       disp        hp       drat         wt      qsec       vs       am      gear      carb
       19.000   118.000    4213.90  2971.000    68.2100    58.9580   333.580    8.000    8.000    72.000    59.000
cyl   118.000   788.000   29003.20 20236.000   416.6800   380.1520  2027.120   36.000   46.000   440.000   394.000
disp 4213.900 29003.200 1106754.59 755107.200 14746.7500 13870.6878 72006.431 1157.300 1441.100 15430.000 13976.500
hp   2971.000 20236.000  755107.20 557527.000 10575.3200  9621.9340 50101.150  802.000 1309.000 11389.000 10906.000
drat   68.210   416.680   14746.75 10575.320   247.9241   209.5513  1198.372   30.040   30.910   262.220   213.410
wt     58.958   380.152   13870.69  9621.934   209.5513   189.2571  1031.586   22.073   21.618   219.295   188.956
qsec  333.580  2027.120   72006.43 50101.150  1198.3716  1031.5862  5936.199  154.760  132.190  1255.630   989.360
vs      8.000    36.000    1157.30   802.000    30.0400    22.0730   154.760    8.000    3.000    31.000    15.000
am      8.000    46.000    1441.10  1309.000    30.9100    21.6180   132.190    3.000    8.000    36.000    31.000
gear   72.000   440.000   15430.00 11389.000   262.2200   219.2950  1255.630   31.000   36.000   284.000   236.000
carb   59.000   394.000   13976.50 10906.000   213.4100   188.9560   989.360   15.000   31.000   236.000   241.000

$val$Xty
          [,1]
       370.700
cyl   2187.800
disp 76118.630
hp   53841.300
drat  1342.342
wt    1112.482
qsec  6589.653
vs     183.900
am     167.100
gear  1426.500
carb  1086.200

$val$XtX
              cyl       disp        hp       drat         wt       qsec      vs      am      gear     carb
       13.000    80.000    3169.20   1723.00    46.8800    43.9940   237.5800   6.000   5.000    46.000   31.000
cyl    80.000   536.000   22869.20  11968.00   274.7200   299.2520  1448.4400  28.000  20.000   270.000  210.000
disp 3169.200 22869.200 1072872.88 536257.20 10348.0460 13220.8010 56795.0730 697.100 424.800 10220.300 9239.600
hp   1723.000 11968.000  536257.20 276751.00  5796.9600  6849.8100 30991.0100 477.000 340.000  5723.000 4870.000
drat   46.880   274.720   10348.05   5796.96   174.8666   149.1677   858.5424  23.990  21.740   170.730  107.850
wt     43.994   299.252   13220.80   6849.81   149.1677   171.6439   796.5083  14.485   9.725   147.287  121.546
qsec  237.580  1448.440   56795.07  30991.01   858.5424   796.5083  4357.2814 115.910  93.490   841.830  558.310
vs      6.000    28.000     697.10    477.00    23.9900    14.4850   115.9100   6.000   4.000    23.000   10.000
am      5.000    20.000     424.80    340.00    21.7400     9.7250    93.4900   4.000   5.000    21.000    7.000
gear   46.000   270.000   10220.30   5723.00   170.7300   147.2870   841.8300  23.000  21.000   168.000  106.000
carb   31.000   210.000    9239.60   4870.00   107.8500   121.5460   558.3100  10.000   7.000   106.000   93.000

$val$Xty
           [,1]
       272.2000
cyl   1505.8000
disp 52586.4500
hp   30521.4000
drat  1037.9350
wt     797.2711
qsec  5025.0920
vs     159.9000
am     150.0000
gear  1010.4000
carb   555.7000
```


Dans **$key**, nous disposons du vecteur (1, 1, 2, 2), nous remarquons que les clés se répètent 2 fois parce que notre fonction reduce() a retourné 2 éléments (X^tX) et (X^ty).

Dans **$val**, nous avons une structure de liste où les matrices (XtX) et (Xty) se succèdent pour chaque valeur de la clé. Pour former la matrice (X^tX) globale [resp. (X^ty)], il faudrait additionner les éléments en position (1, 3) [resp. (2, 4)].

Consolidation des résultats. Les procédures de consolidation suivantes sont opérationnelles quel que soit le nombre de nœuds sollicités (c.-à-.d. nombre de clés K ≥ 1).

```
#consolidation

#X'X
MXtX <- matrix(0,nrow=ncol(mtcars),ncol=ncol(mtcars))
for (i in seq(1,length(resultat$val)-1,2)){
  MXtX <- MXtX + resultat$val[[i]]
}
print(MXtX)

#X'y
MXty <- matrix(0,nrow=ncol(mtcars),ncol=1)
for (i in seq(2,length(resultat$val),2)){
  MXty <- MXty + resultat$val[[i]]
}
print(MXty)
```

Nous obtenons les matrices globales (X^tX) et (X^ty) :

```
> print(MXtX)
                  cyl        disp         hp       drat         wt       qsec       vs       am       gear       carb
        32.000  198.000     7383.10    4694.00   115.0900   102.9520    571.160   14.000   13.000    118.000     90.000
cyl    198.000 1324.000    51872.40   32204.00   691.4000   679.4040   3475.560   64.000   66.000    710.000    604.000
disp  7383.100 51872.400 2179627.47 1291364.40 25094.7960 27091.4888 128801.504 1854.400 1865.900 25650.300 23216.100
hp    4694.000 32204.000 1291364.40  834278.00 16372.2800 16471.7440  81092.160 1279.000 1649.000 17112.000 15776.000
drat   115.090   691.400   25094.80   16372.28   422.7907   358.7190   2056.914   54.030   52.650    432.950    321.260
wt     102.952   679.404   27091.49   16471.74   358.7190   360.9011   1828.095   36.558   31.343    366.582    310.502
qsec   571.160  3475.560  128801.50   81092.16  2056.9140  1828.0946  10293.480  270.670  225.680   2097.460   1547.670
vs      14.000    64.000    1854.40    1279.00    54.0300    36.5580    270.670   14.000    7.000     54.000     25.000
am      13.000    66.000    1865.90    1649.00    52.6500    31.3430    225.680    7.000   13.000     57.000     38.000
gear   118.000   710.000   25650.30   17112.00   432.9500   366.5820   2097.460   54.000   57.000    452.000    342.000
carb    90.000   604.000   23216.10   15776.00   321.2600   310.5020   1547.670   25.000   38.000    342.000    334.000
```

```
> print(MXty)
           [,1]
        642.900
cyl    3693.600
disp 128705.080
hp    84362.700
drat   2380.277
wt     1909.753
qsec  11614.745
vs      343.800
am      317.100
gear   2436.900
carb   1641.900
```

Estimation des paramètres de la régression. Les estimateurs â sont produits à l'aide de procédure **solve**() de R.

```
#coefficients de la régression
a.chapeau <- solve(MXtX,MXty)
print(a.chapeau)
```

A l'issue des calculs, les coefficients de la régression pour les données « mtcars » sont :

```
> print(a.chapeau)
            [,1]
(intercept) →  12.30337416
cyl  -0.11144048
disp  0.01333524
hp   -0.02148212
drat  0.78711097
wt   -3.71530393
qsec  0.82104075
vs    0.31776281
am    2.52022689
gear  0.65541302
carb -0.19941925
```

Vérification - Procédure lm() de R. A titre de vérification, nous avons effectué la régression à l'aide de la procédure lm() de R.

```
> #vérification
> print(summary(lm(mpg ~ ., data = mtcars)))

Call:
lm(formula = mpg ~ ., data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-3.4506 -1.6044 -0.1196  1.2193  4.6271

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 12.30337   18.71788   0.657   0.5181
cyl         -0.11144    1.04502  -0.107   0.9161
disp         0.01334    0.01786   0.747   0.4635
hp          -0.02148    0.02177  -0.987   0.3350
drat         0.78711    1.63537   0.481   0.6353
wt          -3.71530    1.89441  -1.961   0.0633 .
qsec         0.82104    0.73084   1.123   0.2739
vs           0.31776    2.10451   0.151   0.8814
am           2.52023    2.05665   1.225   0.2340
gear         0.65541    1.49326   0.439   0.6652
carb        -0.19942    0.82875  -0.241   0.8122
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.65 on 21 degrees of freedom
Multiple R-squared:  0.869, Adjusted R-squared:  0.8066
F-statistic: 13.93 on 10 and 21 DF,  p-value: 3.793e-07
```

Les paramètres estimés concordent en tous points. Ouf, j'aurais été bien embêté sinon.

6 Conclusion

Des exemples très scolaires ont été mis en avant dans ce tutoriel pour illustrer la programmation MapReduce à l'aide du package « rmr2 » sous R. L'idée directrice est la subdivision des calculs sur un groupe (cluster) de machines (nœuds). Bien sûr, d'autres solutions existent. J'avais moi-

même exploré la parallélisation des tâches en utilisant d'autres packages[13]. Je m'étais placé dans le cadre de l'exploitation efficace des machines multi-cœurs. Mais l'extension à la répartition des calculs sur plusieurs machines distantes était prévue par les librairies étudiées.

Je parle d'exemples « scolaires » parce que les données arrivent en un seul bloc dans notre étude. La fonction « map » est appelée une seule fois et se charge de dispatcher les données. Pour aller plus loin, il faudrait se placer sur une configuration où les données arrivent par blocs – par exemple en provenance de différentes machines – occasionnant plusieurs appels à la fonction map qui les réorganise avant de passer la main à la fonction reduce, qui peut être appelée plusieurs fois ou non selon le nombre de valeurs distinctes de la clé. Ceci serait possible par exemple si l'on travaillait dans un véritable environnement Hadoop avec un cluster à plusieurs nœuds. Bref, il reste encore de la place pour plusieurs tutoriels à venir…

[13] Tutoriel Tanagra, « Programmation parallèle sous R », juin 2013.