1 Introduction

Mise en œuvre des algorithmes de descente de gradient sous R. Utilisation des packages « gradDescent » et « tensorflow / keras ».

Ce tutoriel fait suite au support de cours consacré à l'application de la méthode du gradient en apprentissage supervisé (RAK, 2018). Nous travaillons sous R. Un document consacré à Python viendra par la suite.

Nous nous plaçons dans le cadre de la régression linéaire multiple. Dans un premier temps, nous traiterons un jeu de données réduit qui nous permettra d'étudier en détail le comportement des algorithmes de descente de gradient, stochastique ou non. L'idée est de comparer les coefficients estimés et les valeurs de la fonction de perte obtenues à l'issue du processus d'apprentissage. Dans un second temps, nous traiterons un fichier réaliste de classement de protéines où le nombre de variables est élevé, son ratio par rapport au nombre d'observations est largement supérieur à 1. Dans ce cas, l'implémentation usuelle de la régression sous R, **lm()** du package « stats », même si elle est solide, n'est pas opérationnelle. Seules les approches basées sur la descente de gradient permettent de produire un résultat exploitable.

Nous utiliserons les packages 'gradDescent' et 'tensorflow / keras'. Ce dernier tandem a été présenté plus en détail dans un précédent document (Avril 2018). Il faut s'y référer notamment pour la partie installation qui n'est pas triviale.

2 Un premier exemple : étude des coefficients

2.1 Données

Le fichier "**toy_data.txt**" est composé de 50 observations, une variable cible "y", et 10 variables prédictives candidates (x1, …, x10). Ayant été générées artificiellement, nous savons que seules x4 et x7 sont pertinentes pour la prédiction.

Nous importons et inspectons les données dans un premier temps :

```
#modification du dossier de travail
setwd("… votre dossier de travail …")

#importation des données
DToy <- read.table("toy_data.txt",sep="\t",header=T)
print(str(DToy))
```

Nous obtenons la liste des variables :

```
'data.frame':50 obs. of  11 variables:
 $ x1 : num  -0.306 -0.208 -0.406 -0.28 -0.018 -0.466 0.066 0.329 -0.117...
 $ x2 : num  0.274 -0.394 0.15 -0.042 -0.136 0.386 -0.47 -0.398 0.366 0.281 ...
 $ x3 : num  -0.336 0.24 -0.397 0.185 0.04 0.237 -0.001 0.429 0.499 0.287 ...
 $ x4 : num  0.331 -0.262 -0.106 -0.182 -0.193 0.214 -0.034 -0.126 0.057 0.494 ...
 $ x5 : num  -0.117 0.17 0.06 -0.282 -0.421 0.086 0.388 -0.071 0.382 -0.197 ...
 $ x6 : num  0.294 -0.109 0.291 -0.268 -0.301 0.029 0.331 0.481 -0.491 0.251 ...
 $ x7 : num  0.337 -0.29 0.109 0.258 -0.341 0.301 0.273 0.172 -0.36 -0.263 ...
 $ x8 : num  -0.134 0.167 -0.067 0.379 0.268 0.286 0.056 -0.327 0.356 -0.295 ...
 $ x9 : num  0.464 -0.379 -0.181 -0.313 0.251 0.231 -0.201 0.148 -0.213 -0.057 ...
 $ x10: num  0.24 -0.292 -0.326 -0.232 0.059 -0.275 -0.213 0.462 0.226 0.112 ...
 $ y  : num  4.46 -1.86 1.78 2.53 -2.11 ...
```

2.2 Critère MSE

L'objectif de la régression est de prédire au mieux les valeurs de "y" à partir des variables (x1, …, x10). Formellement, elle s'écrit :

$$y = a_0 + a_1 x_1 + \cdots + a_p x_p + \varepsilon$$

Le terme d'erreur ε résume les insuffisances du modèle. Le critère des moindres carrés vise à minimiser la quantité :

$$\sum_i \varepsilon_i^2$$

En pratique, sur un échantillon de taille n, nous utilisons la MSE (mean squared error, moyenne des carrés des résidus) pour évaluer la qualité de la prédiction :

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

Où y_i et $\hat{y}_i$ sont respectivement les valeurs observées et prédites par le modèle pour l'observation n°i, n est la taille de l'échantillon.

Dans le meilleur des cas, MSE = 0, les prédictions sont parfaites.

Mais quelle serait la valeur du MSE dans le pire des cas, lorsque les prédictives « x_j » n'amènent aucune information sur « y » ? On montre facilement que dans la régression avec la constante (a_0) comme seul paramètre, la prédiction optimale est la moyenne de « y », soit $\hat{a}_0 = \bar{y}$. Nous pouvons dès lors calculer le MSE correspondant :

```
# predicteur par défaut, moyenne de y
default_pred <- mean(DToy$y)

#calcul du MSE correspondant
```

```
default_mse <- mean((DToy$y-default_pred)^2)
print(default_mse)
```

Elle est égale à **6.348566**.

A priori, les différents modèles développés par la suite devraient faire mieux en termes de MSE.

2.3 Régression linéaire avec lm()

Sur un aussi petit fichier, nous pouvons obtenir la solution optimale (correspondant au minimum global de la MSE) à l'aide de la régression linéaire multiple usuelle. Nous utilisons la fonction **lm()** du package « stats », installé et chargé automatiquement par R.

```
#régression avec lm
regToy <- lm(y ~ ., data = DToy)

#résultats détaillés
sregToy <- summary(regToy)
print(sregToy)
```

```
Call:
lm(formula = y ~ ., data = DToy)

Residuals:
     Min       1Q   Median       3Q      Max
-0.19647 -0.07781  0.01052  0.06424  0.15071

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.002399   0.016569  60.497   <2e-16 ***
x1           0.044944   0.059085   0.761   0.4514
x2           0.072616   0.057193   1.270   0.2117
x3          -0.020233   0.057190  -0.354   0.7254
x4           1.882630   0.057365  32.819   <2e-16 ***
x5           0.106408   0.053350   1.995   0.0531 .
x6           0.030316   0.051401   0.590   0.5587
x7           8.033472   0.049291 162.980   <2e-16 ***
x8          -0.050131   0.053514  -0.937   0.3546
x9           0.019780   0.060367   0.328   0.7449
x10          0.009624   0.052959   0.182   0.8567
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.09978 on 39 degrees of freedom
Multiple R-squared:  0.9988,     Adjusted R-squared:  0.9985
F-statistic:  3185 on 10 and 39 DF,  p-value: < 2.2e-16
```

Seuls les coefficients associés aux variables « x4 » et « x7 » sont significatifs. C'était attendu.

Surtout, nous avons là les "vrais" coefficients que nous devrons retrouver avec les autres approches. Calculons le MSE de la régression.

```
#MSE pour la régression avec lm()
lm_mse <- mean((DToy$y - regToy$fitted.values)^2)
```

```
print(lm_mse)
```

Elle est égale à **MSE = 0.007765202**. C'est la valeur optimale que nous devons obtenir. Voyons ci-dessous dans quelle mesure les algorithmes basés sur la descente de gradient permettent de s'en approcher.

2.4 Descente de gradient

2.4.1 Descente de gradient – Version 1

Nous utilisons la librairie « gradDescent » dans cette section. Après l'avoir installé, nous la chargeons et nous faisons appel à la fonction **gradDescentR.learn()**.

```
#chargement de la librairie
library(gradDescent)

#appel de la fonction - construction du modèle
mgd1 <- gradDescentR.learn(DToy,featureScaling=FALSE,learningMethod="GD",seed=1)
```

Nous passons en paramètres :

- Les données à traiter, la variable cible doit être en dernière position.
- Nos variables sont sur la même échelle, aucune transformation n'est effectuée (featureScaling = FALSE). Si (featureScaling=TRUE), les variables sont centrées et réduites.
- Nous demandons l'algorithme de descente de gradient usuelle (learningMethod = 'GD') c.-à-d. le vecteur gradient est calculé sur la totalité des individus avant de venir corriger les coefficients estimés.
- seed permet de fixer la valeur de départ du générateur de nombre aléatoire afin d'obtenir le même résultat à chaque exécution.
- Il n'apparaît pas explicitement ici, mais le nombre maximum d'itération est (maxIter = 10). Il constitue également la règle d'arrêt du processus.

Nous faisons afficher les paramètres estimés du modèle.

```
#affichage du modèle
print(mgd1$model)
```

```
          [,1]      [,2]     [,3]      [,4]      [,5]      [,6]
[1,] 0.7706671 0.2535502 0.410788 0.9425837 0.3145219 0.8492295
          [,7]     [,8]      [,9]      [,10]     [,11]
[1,] 0.9954069 1.324273 0.6607114 0.05475434 0.1380973
```

La comparaison avec ceux de **lm()** (section 2.3) n'est pas aisée en l'état. Nous créons une petite fonction de mise en forme.

```
#fonction pour la mise en forme des coefficients
print.coefs <- function(coefs,D){
  coefs1 <- matrix(coefs,ncol=1,nrow=length(coefs))
  rownames(coefs1) <- c("intercept",colnames(D)[-ncol(D)])
  print(coefs1)
}

#affichage des coefficients de mgd1
print.coefs(mgd1$model,DToy)
```

Nous distinguons les paramètres pour chaque variable :

```
                [,1]
intercept 0.77066712
x1        0.25355024
x2        0.41078797
x3        0.94258368
x4        0.31452185
x5        0.84922950
x6        0.99540687
x7        1.32427261
x8        0.66071138
x9        0.05475434
x10       0.13809732
```

A vue d'œil, les coefficients sont très différents, cela ne laisse pas augurer d'une bonne qualité de modélisation. Mesurons justement la MSE. Nous créons à cet effet une fonction ad hoc prenant en entrée les données à traiter et les coefficients du modèle.

```
#calcul du mse
calc.mse <- function(D,coefs){
  #prédiction : application des coefficients estimés sur les données
  yc <- apply(as.matrix(cbind(1,D[-ncol(D)])),1,function(x){sum(x*coefs)})
  #résidu
  res <- D[ncol(D)]-yc
  #mse
  return(mean(res^2))
}

#MSE pour mgd1
print(calc.mse(DToy,mgd1$model))
```

Nous obtenons **MSE = 4.463887**. Même si nous faisons mieux que le modèle par défaut (6.348566), nous sommes très loin du compte (0.007765202 pour lm). Il va falloir trouver des solutions pour améliorer les résultats.

2.4.2 Descente de gradient – Version 2

Le paramétrage est essentiel pour ce type d'heuristique. Une solution simple pour améliorer les résultats consiste à augmenter le nombre d'itérations. Après plusieurs tentatives, nous arrivons à maxIter = 1000.

```
#passer le nombre d'itérations à maxIter = 1000
mgd2 <- gradDescentR.learn(DToy,featureScaling = FALSE,learningMethod="GD", seed = 1, control=list(maxIter=1000))

#calcul et affichage du mse
print(calc.mse(DToy,mgd2$model))
```

La MSE passe à **0.007866008**. C'est autrement plus encourageant, nous sommes proches de la valeur optimale (0.007765202 pour lm). Affichons les coefficients estimés.

```
#coefficients du second modèle
print.coefs(mgd2$model,DToy)
```

```
                [,1]
intercept  0.99760103
x1         0.01249382
x2         0.06755182
x3         0.01007553
x4         1.88268525
x5         0.11009560
x6         0.04313524
x7         8.01895593
x8        -0.04716026
x9         0.03579809
x10        0.01329515
```

A peu de choses près, nous retrouvons les coefficients de **lm()** (section 2.3), notamment ceux de x4 et X7 qui étaient les seules variables pertinentes.

2.4.3 Etude de la fonction

Par curiosité, j'ai récupéré le code source de la fonction de descente de gradient du package "gradDescent".

```
#fonction descente de gradient du package gradDescent
GD <- function(dataTrain, alpha=0.1, maxIter=10, seed=NULL){
    #convert data.frame dataSet in matrix
    dataTrain <- matrix(unlist(dataTrain), ncol=ncol(dataTrain), byrow=FALSE)
    #shuffle data train
    set.seed(seed)
    dataTrain <- dataTrain[sample(nrow(dataTrain)), ]
    set.seed(NULL)
    #initialize theta
    theta <- getTheta(ncol(dataTrain), seed=seed)
    #bind 1 column to dataTrain
    dataTrain <- cbind(1, dataTrain)
    #parse dataTrain into input and output
```

```
    inputData <- dataTrain[,1:ncol(dataTrain)-1]
    outputData <- dataTrain[,ncol(dataTrain)]
    #temporary variables
    temporaryTheta <- matrix(ncol=length(theta), nrow=1)
    #updateRule <- matrix(0, ncol=length(theta), nrow=1)
    #constant variables
    rowLength <- nrow(dataTrain)
    #loop the gradient descent
    for(iteration in 1:maxIter){
        error <- (inputData %*% t(theta)) - outputData
        for(column in 1:length(theta)){
            term <- error * inputData[,column]
            #calculate gradient
            gradient <- sum(term) / rowLength
            temporaryTheta[1,column] = theta[1,column] - (alpha*gradient)
        }
        #update all theta in the current iteration
        theta <- temporaryTheta
    }
    result <- theta
    return(result)
}
```

Ce code correspond à une traduction point par point de l'algorithme du gradient.

Nous apprenons quand même plusieurs choses sur les choix d'implémentation :

- La valeur par défaut du taux d'apprentissage est alpha = 0.1 (il est noté η dans mon support de cours ; RAK, 2018). Il est fixe tout au long du processus. Il peut être décroissant graduellement au cours du processus dans d'autres packages.
- L'algorithme réalise exactement le nombre d'itérations demandé. Il n'y a pas de possibilité de sortie prématurée de la boucle principale ***for***. On aurait pu imaginer d'autres règles d'arrêt comme la comparaison entre deux vecteurs de coefficients successifs, ou encore la stagnation des valeurs de la fonction de perte à optimiser.
- Ces dernières d'ailleurs ne sont pas disponibles. Il n'est pas possible d'afficher une courbe retraçant leur évolution au fil des itérations.

Même si je les scrute toujours, je parle rarement des codes source des packages dans mes tutoriels. Les choix de programmation jouent souvent sur la robustesse et la rapidité d'exécution, moins sur la qualité du résultat obtenu. Je me vois mal commencer à pinailler sur tel ou tel aspect des programmes développés par d'autres chercheurs. Je déroge à cette règle ici parce que nous verrons que leur étude donnera un éclairage édifiant sur le comportement de l'algorithme dans la section suivante.

2.5 Descente de gradient stochastique (SGD)

La descente de gradient stochastique (SGD – stochastic gradient descent) met à jour itérativement les paramètres estimés lors du traitement d'un (online) ou de plusieurs (mini-batch) individus. L'approche se démarque surtout lors du traitement de très grandes bases de données, lorsque le calcul du gradient sur l'ensemble des observations devient coûteux en temps de calcul.

Ce n'est pas vraiment le cas pour notre jeu de données (50 individus, 10 variables). Mais, néanmoins, l'approche devrait faire montre de son efficacité, même pour les problèmes faciles.

2.5.1 SGD – Version 1

Nous nous tournons de nouveau vers la librairie « gradDescent ». Nous lançons les calculs avec les paramètres par défaut, sans transformation de variables en demandant l'optimisation à l'aide de la descente de gradient stochastique (learningMethod = "SGD") :

```
#descente de gradient stochastique
msgd1 <- gradDescentR.learn(DToy,featureScaling=FALSE,learningMethod="SGD",seed=1)

#mse
print(calc.mse(DToy,msgd1$model))
```

Avec le même paramétrage par défaut (maxIter = 10), la descente de gradient stochastique fait pire (**MSE = 5.256616**) que la descente de gradient usuelle (MSE = 4.463887 ; section 2.4.1). Ça me paraissait très étrange. Nous aurions dû avoir un résultat au moins similaire.

Comme nous avons le code source à disposition, voyons ce qu'il en est.

```
#stochastic gradient descent
SGD <- function(dataTrain, alpha=0.1, maxIter=10, seed=NULL){
    #convert data.frame dataSet in matrix
    dataTrain <- matrix(unlist(dataTrain), ncol=ncol(dataTrain), byrow=FALSE)
    #shuffle dataTrain
    set.seed(seed)
    dataTrain <- dataTrain[sample(nrow(dataTrain)), ]
    set.seed(NULL)
    #initialize theta
    theta <- getTheta(ncol(dataTrain), seed=seed)
    #bind 1 column to dataTrain
    dataTrain <- cbind(1, dataTrain)
    #parse dataTrain into input and output
    inputData <- dataTrain[,1:ncol(dataTrain)-1]
    outputData <- dataTrain[,ncol(dataTrain)]
    #temporary variables
    temporaryTheta <- matrix(ncol=length(theta), nrow=1)
    # updateRule <- matrix(0, ncol=length(theta), nrow=1)
```

```
    #constant variables
    rowLength <- nrow(dataTrain)
    set.seed(seed)
    stochasticList <- sample(1:rowLength, maxIter, replace=TRUE)
    set.seed(NULL)
    #loop the gradient descent
    for(iteration in 1:maxIter){
        error <- (inputData[stochasticList[iteration],] %*% t(theta)) - outputData[stochasticList[iteration]]
        for(column in 1:length(theta)){
            #calculate gradient
            gradient <- error * inputData[stochasticList[iteration], column]
            temporaryTheta[1,column] = theta[1,column] - (alpha*gradient)
        }
        #update all theta in the current iteration
        theta <- temporaryTheta
    }
    result <- theta
    return(result)
}
```

La fonction implémente la version “online” c.-à-d. le gradient est calculé, et donc les coefficients estimés sont mis à jour, au passage de chaque observation.

En revanche, nous notons une vraie curiosité. Le programme extrait au hasard maxIter observations, avec remise, et réalise les calculs exclusivement sur ces individus (stochasticList). Avec le paramétrage par défaut (maxIter = 10), nous n’exploitons que 10 individus sur 50 (peut-être moins puisque c’est un tirage avec remise), pour réaliser le processus de modélisation. Finalement, que notre exécution ci-dessus soit moins efficace que la descente de gradient usuelle se comprend parfaitement.

Je ne voyais pas vraiment les choses ainsi. Il faut effectivement mettre à jour les coefficients au passage de chaque individu, qui doivent être également mélangés au hasard pour éviter que leur ordre n’influe sur les résultats. Mais le nombre d’itérations correspondrait plutôt au nombre de passage de la base entière à mon sens. On peut faire le parallèle à la notion d’epochs que l’on trouve dans les implémentations des réseaux de neurones.

En l’état, pour pouvoir exploiter pleinement cette fonction, il faudrait fixer une valeur très élevée de maxIter, quelque chose qui ressemblerait à « n x epochs », où n est la taille de la base.

A ce stade, il m’a paru plus judicieux d’explorer une autre implémentation de la SGD.

2.5.2 SGD – Version 2 – Librairies « Tensorflow / Keras »

La librairie "keras" est une surcouche de "tensorflow" qui permet d'accéder facilement aux algorithmes de deep learning. Pour réaliser la régression linéaire multiple, nous construisons un perceptron simple avec une fonction d'activation linéaire.

L'installation des librairies « tensorflow » et « keras » requiert des manipulations supplémentaires décrites dans un précédent tutoriel (http://tutoriels-data-mining.blogspot.fr/2018/04/deep-learning-tensorflow-et-keras-sous-r.html).

Nous créons un nouveau réseau avec la commande **keras_model_sequential()**. Nous y insérons une seule couche (**layer_dense**) que fait directement la connexion entre l'entrée et la sortie, cette dernière comportant un seul neurone puisqu'il s'agit de prédire les valeurs de "y". Nous utilisons une fonction d'activation linéaire (activation = 'linear') parce que nous réalisons une régression linéaire ; 'input_shape=c(10)' correspond au nombre de variables d'entrées.

```
#importation de la librairie keras
library(keras)

#structure de réseau de neurones
msgdk1 <- keras_model_sequential()

#une couche
msgdk1 %>%
  layer_dense(units=1,input_shape = c(10),activation="linear")
```

Affichons l'architecture du réseau pour vérification.

```
#architecture du réseau
print(summary(msgdk1))
```

```
_____________________________________________________________________
Layer (type)                 Output Shape              Param #
=====================================================================
dense_1 (Dense)              (None, 1)                 11
=====================================================================
Total params: 11
Trainable params: 11
Non-trainable params: 0
_____________________________________________________________________
```

Il y a bien 11 paramètres à estimer en incluant la constante (intercept) de la régression.

L'instruction **compile()** spécifie la fonction à optimiser (loss), nous optons pour la MSE, et l'algorithme d'optimisation (optimizer), à savoir la descente de gradient stochastique.

```
#compilation
msgdk1 %>% compile(
```

```
  loss="mean_squared_error",
  optimizer=optimizer_sgd(),
  metrics="mean_squared_error"
)
```

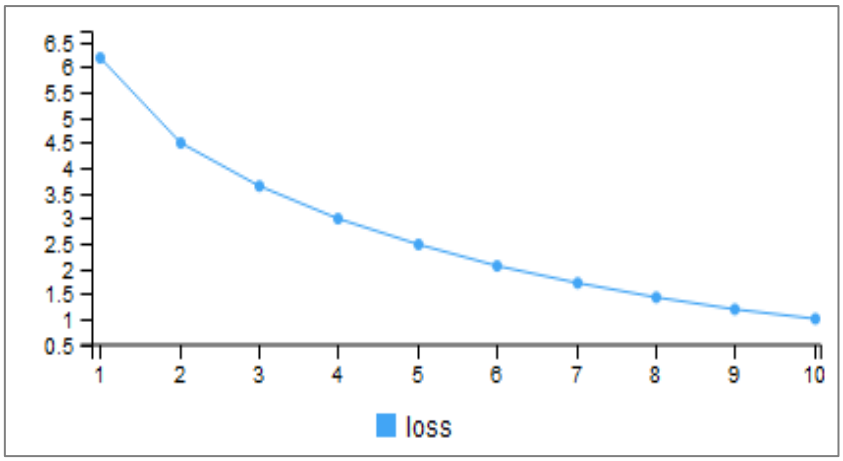
Il ne reste plus qu'à lancer le processus d'apprentissage avec **fit()**. Nous faisons passer individuellement les individus (batch_size = 1), avec 10 itérations (epochs = 10) c.-à-d. la base entière est passé 10 fois.

```
#construction du modèle
msgdk1 %>% fit(
  x=as.matrix(DToy[,-ncol(DToy)]),
  y=DToy[,ncol(DToy)],
  epochs = 10,
  batch_size = 1
)
```

Dans la console R apparaît le déroulement du processus d'optimisation :

```
2018-04-25 15:21:54.513017: I C:\tf_jenkins\workspace\rel-win\M\windows\PY\36\tensorflow\core\platform\cpu_feature_guard.cc:137] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX AVX2
Epoch 1/10
50/50 [==============================] - 1s 12ms/step - loss: 6.1835 - mean_squared_error: 6.1835
Epoch 2/10
50/50 [==============================] - 0s 751us/step - loss: 4.4936 - mean_squared_error: 4.4936
Epoch 3/10
50/50 [==============================] - 0s 771us/step - loss: 3.6373 - mean_squared_error: 3.6373
Epoch 4/10
50/50 [==============================] - 0s 701us/step - loss: 2.9900 - mean_squared_error: 2.9900
Epoch 5/10
50/50 [==============================] - 0s 742us/step - loss: 2.4766 - mean_squared_error: 2.4766
Epoch 6/10
50/50 [==============================] - 0s 756us/step - loss: 2.0514 - mean_squared_error: 2.0514
Epoch 7/10
50/50 [==============================] - 0s 735us/step - loss: 1.7120 - mean_squared_error: 1.7120
Epoch 8/10
50/50 [==============================] - 0s 770us/step - loss: 1.4272 - mean_squared_error: 1.4272
Epoch 9/10
50/50 [==============================] - 0s 727us/step - loss: 1.1892 - mean_squared_error: 1.1892
Epoch 10/10
50/50 [==============================] - 0s 768us/step - loss: 1.0006 - mean_squared_error: 1.0006
```

Le suivi est aussi disponible sous la forme d'un graphique.

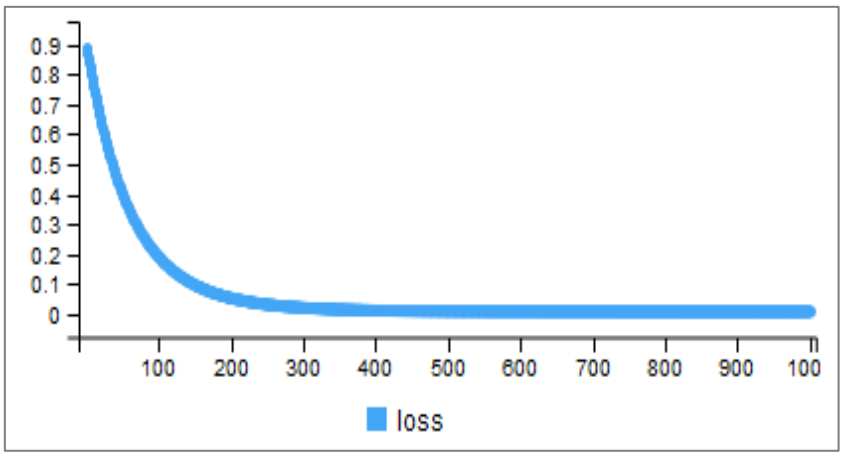


Nous sommes loin de la valeur optimale à l'issue du process (**MSE = 1.0006**). Mais à effort de calcul égal (10 itérations sur la base complète), nous faisons nettement mieux que la descente de gradient usuelle (MSE = 4.463887 ; section 2.4.1).

Nous augmentons le nombre d'itérations (epochs = 1000) en prenant les individus par groupes de 10 (stratégie mini-batch, batch_size = 10). En réalité, il s'agit de 1000 itérations supplémentaires par rapport à la dernière exécution. En effet, **fit()** commence à partir des valeurs dernièrement estimées des coefficients.

```
#1000 itérations supplémentaires
msgdk1 %>% fit(
  x=as.matrix(DToy[,-ncol(DToy)]),
  y=DToy[,ncol(DToy)],
  epochs = 1000,
  batch_size = 10
)
```

On aurait pu se dispenser d'autant de calculs. Nous stagnons dès la 300ème itération.



Nous obtenons au final **MSE = 0.0078**, très proche de l'optimum de la régression avec **lm()**.

Les coefficients estimés sont de facto similaires (voir section 2.3), notamment en ce qui concerne ceux des variables pertinentes x4 et x7.

```
#paramètres estimés du modèle
print(get_weights(msgdk1))
```

```
[[1]]
             [,1]
 [1,]  0.03172033
 [2,]  0.06813724
 [3,] -0.01000195
 [4,]  1.88443446
 [5,]  0.10428251
 [6,]  0.03575632
 [7,]  8.02612019
 [8,] -0.04902204
 [9,]  0.02266809
[10,]  0.01184312

[[2]]
[1] 1.000765
```

Nous avons d'abord les coefficients dans l'ordre des variables, puis l'intercept.

3 Un exemple réaliste : discrimination de familles de protéines

3.1 Problème et données

Nous nous plaçons dans le cadre du text mining pour ce second exemple.

Une protéine est représentée par une suite de caractères pris dans un alphabet de 20 signes (les acides aminés). L'analogie entre un texte et une séquence de protéine est raisonnable, à la différence qu'il n'existe pas dans ce cas de séparateurs naturels comme l'espace ou la ponctuation entre les groupes de caractères. Par exemple :

```
<description>
SQFRVSPLDRTWNLGETVELKCQVLLSNPTSGCSWLFQPRGAAASPTFLLYLSQNKPKAAEGLDTQRFSGKRLGDTFVLTLSD
FRRENEGYYFCSALSNSIMYFSHFVPVFLPA
</description>
```

Les protéines sont classées en familles selon leur fonction. Il est admis que les protéines appartenant à la même famille ont des structures similaires. L'objectif est de discriminer deux familles de protéines à partir de leur description primaire.

Les pré-traitements ont été réalisées en amont. Les descripteurs sont des 3-grams (suite de 3 caractères) extraits à partir de la représentation des protéines. Notre fichier comporte 101 observations et 7143 descripteurs avec une pondération binaire. La variable cible « y » est binaire (0/1), placée en dernière position. Elle distingue les 2 familles à reconnaître. Ce fichier a déjà été utilisé dans un ancien tutoriel. J'avais montré que les propriétés de régularisation de la

régression PLS (partial least square) faisaient merveille dans ce contexte de très forte dimensionnalité où le ratio entre le nombre de variables et d'observations est inversé par rapport aux situations usuelles.

Nous importons le fichier dans un premier temps.

```
#modification du dossier de travail
setwd("… votre répertoire des données …")

#importation des données
DProt <- read.table("protein01.txt",sep="\t",header=T)
print(dim(DProt))
```

Nous disposons de 101 observations et 7144 variables (prédictives + cible).

Nous fractionnons les données en échantillons d'apprentissage (51 obs.) et test (50 obs.) à l'aide d'un index généré aléatoirement[1].

```
#index pour la partition apprentissage-test
set.seed(1)
idTrain <- sample(1:nrow(DProt),51)
```

L'idée est de développer un modèle sur l'échantillon d'apprentissage, qui sera le plus efficace possible sur l'échantillon test. Si tant est qu'il soit possible de produire un modèle qui tient la route dans ce contexte.

Remarque : On peut s'étonner de l'utilisation de la régression dans un problème de classement. C'est tout à fait pertinent en réalité. On montre même que, moyennant certains ajustements, il est possible de retrouver les résultats de l'analyse discriminante linéaire à partir d'un programme de régression linéaire multiple (RAK, 2014).

3.2 Régression linéaire multiple

Nous disposons de 7143 variables explicatives et 51 observations pour l'apprentissage. L'implémentation usuelle de la régression linéaire ne devrait pas y survivre, même s'il s'agit de lm() de R qui s'appuie sur une décomposition QR, très stable, pour réaliser ses calculs.

```
#régression impossible, nombre de variables > nombre d'observations
```

[1] Sur un échantillon d'une taille aussi réduite (101 observations), il est plus judicieux de passer par des techniques de rééchantillonnage telles que la validation croisée ou le bootstrap. Nous simplifions ici, notre propos est de montrer l'intérêt de la descente de gradient lors de l'appréhension des bases à forte dimensionalité.

```
regp <- lm(y ~ ., data = D[idTrain,])
```

Très curieusement, R ne renvoie aucun message d'erreur. J'avoue avoir été dubitatif. Inspectons les coefficients pour savoir ce qu'il en est réellement. Nous affichons les 50 premiers.

```
#affichage des 50 premiers coefficients
print(regp$coefficients[1:50])
```

Je n'ai pas d'expertise sur les valeurs en elles-mêmes. En revanche, nous notons qu'il y a des N/A (not available) pour certaines d'entre-elles.

```
  (Intercept)           EHT           HTY           TYG           YGE           GEV
-2.046026e-01 -9.468456e-01 -7.718122e-01 -5.472430e-01 -2.336761e-01  1.644670e+00
          EVN           VNQ           NQL           QLG           LGG           GGV
 9.568672e-02 -3.750866e-01 -1.571940e+00 -1.172817e+00  1.356845e-01 -3.553860e-01
          GVF           VFV           FVN           VNG           NGR           GRP
 5.472430e-01  7.431900e-01  1.382308e+00  1.485248e+00  5.599886e-01  3.481710e+00
          RPL           PLP           LPN           PNA           NAI           AIR
-1.069700e+00  9.423204e-02 -5.304395e-01  7.715261e-01 -6.661321e-02 -7.009115e-01
          IRL           RLR           LRI           RIV           IVE           VEL
 4.666269e-01 -1.919819e-01 -1.456638e+00 -1.287052e+00 -5.342435e-01 -7.491184e-16
          ELA           LAQ           AQL           LGI           GIR           IRP
-5.763930e-01  7.809956e-01  4.236070e-01 -4.214955e-02  5.173635e-01 -9.729735e-01
          RPC           PCD           CDI           DIS           ISR           SRQ
 4.174095e-01 -3.387478e+00 -3.047622e-01  1.205642e+00 -1.035501e+00 -3.419916e-02
          RQL           QLR           LRV           RVS           VSH           SHG
 6.135257e-01  1.674699e-01 -4.646967e-01  7.848117e-01  2.984720e-02  6.423776e-01
          HGC           GCV
           NA            NA
```

Pour avoir le cœur net, recensons les coefficients qui sont réellement disponibles.

```
#liste des coefficients disponibles
print(regp$coefficients[!is.na(regp$coefficients)])
```

```
  (Intercept)           EHT           HTY           TYG           YGE           GEV
-2.046026e-01 -9.468456e-01 -7.718122e-01 -5.472430e-01 -2.336761e-01  1.644670e+00
          EVN           VNQ           NQL           QLG           LGG           GGV
 9.568672e-02 -3.750866e-01 -1.571940e+00 -1.172817e+00  1.356845e-01 -3.553860e-01
          GVF           VFV           FVN           VNG           NGR           GRP
 5.472430e-01  7.431900e-01  1.382308e+00  1.485248e+00  5.599886e-01  3.481710e+00
          RPL           PLP           LPN           PNA           NAI           AIR
-1.069700e+00  9.423204e-02 -5.304395e-01  7.715261e-01 -6.661321e-02 -7.009115e-01
          IRL           RLR           LRI           RIV           IVE           VEL
 4.666269e-01 -1.919819e-01 -1.456638e+00 -1.287052e+00 -5.342435e-01 -7.491184e-16
          ELA           LAQ           AQL           LGI           GIR           IRP
-5.763930e-01  7.809956e-01  4.236070e-01 -4.214955e-02  5.173635e-01 -9.729735e-01
          RPC           PCD           CDI           DIS           ISR           SRQ
 4.174095e-01 -3.387478e+00 -3.047622e-01  1.205642e+00 -1.035501e+00 -3.419916e-02
          RQL           QLR           LRV           RVS           VSH           SHG
 6.135257e-01  1.674699e-01 -4.646967e-01  7.848117e-01  2.984720e-02  6.423776e-01
          CVS           SLA           TFK
 1.125320e+00 -1.180253e-15 -1.056823e-15
```

Soit 51 coefficients parmi 7144 (en comptant l'intercept) que l'on aurait dû obtenir.

Ce modèle n'est pas exploitable. Nous ne pouvons pas le déployer sur un nouvel individu à classer. La tentation de considérer comme nulles les coefficients non disponibles ne relève d'aucune logique statistique. Force est de constater que nous sommes bloqués à ce stade.

3.3 Descente de gradient stochastique

Modélisation. La descente de gradient stochastique permet de réaliser une régression. Elle s'appuie sur une mécanique de calcul adaptée au traitement des grandes bases, notamment parce que son occupation mémoire est contenue (RAK, 2018). Nous faisons appel au tandem « tensorflow / keras » dans cette section. Nous créons un perceptron simple.

```
#keras
library(keras)

#structure vide de réseau
mgdkp <- keras_model_sequential()

#couche unique reliant l'entrée à la sortie
mgdkp %>%
  layer_dense(units=1,input_shape = c(ncol(DProt)-1),activation="linear")

#vérification de l'architecture
print(summary(mgdkp))
```

Nous avons 7144 paramètres à estimer.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 1)                 7144
=================================================================
Total params: 7,144
Trainable params: 7,144
Non-trainable params: 0
_________________________________________________________________
```

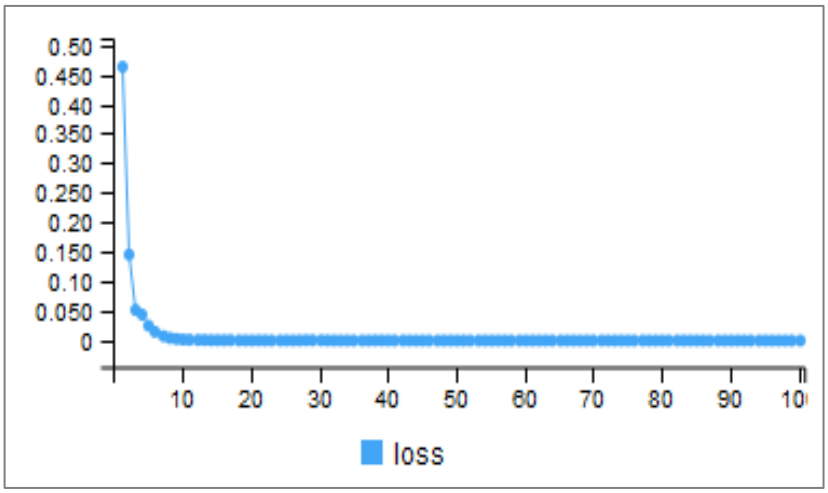
Nous lançons la modélisation sur l'échantillon d'apprentissage après avoir configurer l'algorithme d'optimisation.

```
#compilation
mgdkp %>% compile(
  loss="mean_squared_error",
  optimizer=optimizer_sgd(),
  metrics="mean_squared_error"
)

#apprentissage
mgdkp %>% fit(
  x=as.matrix(DProt[idTrain,-ncol(DProt)]),
```

```
  y=DProt[idTrain,ncol(DProt)],
  epochs = 100,
  batch_size = 32
)
```

Le processus a été mené à bon terme avec une **MSE = 5.0077 x 10^{-5}**, quasi-nulle. L'inspection de la courbe d'évolution de la MSE montre qu'on aurait pu se contenter d'une valeur nettement plus faible de l'epochs.



Etude des coefficients. La mécanique semble avoir fonctionné. Pour s'en assurer, nous étudions les variables. Nous les récupérons dans un vecteur dédié.

```
#récuparation des coefficients
coefs <- get_weights(mgdkp)[[1]][,1]

#associer les coefficients aux noms des variables
names(coefs) <- colnames(DProt)[-ncol(DProt)]

#affichage des coefs des 10 premières variables
print(coefs[1:10])
```

```
         EHT          HTY          TYG          YGE          GEV
-0.005883977 -0.017279705 -0.009658237  0.011849855  0.006385316
         EVN          VNQ          NQL          QLG          LGG
-0.018898567  0.013650944 -0.017897310 -0.018202713  0.008108379
```

Est-ce qu'il a des coefficients N/A ?

```
#nombre de N/A
print(length(which(is.na(coefs))))
```

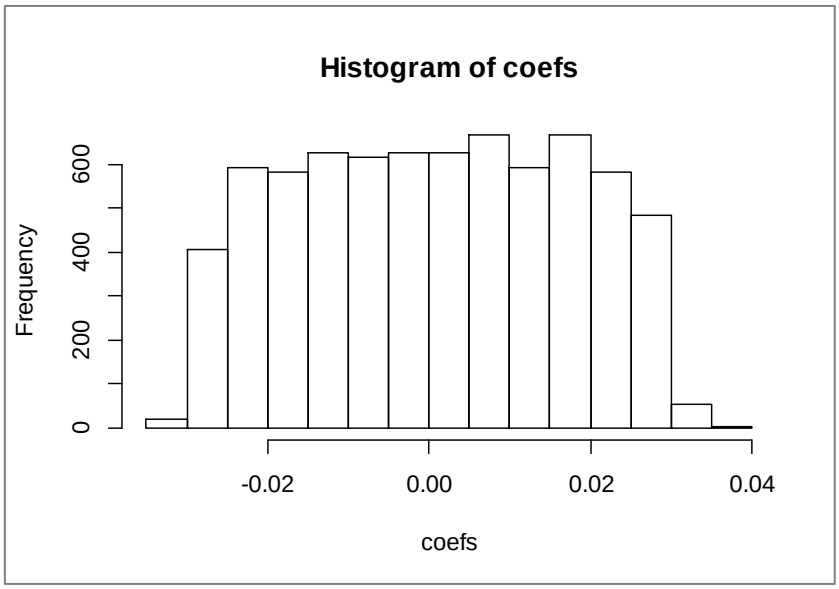
```
0
```

Aucun visiblement. C'est une très bonne nouvelle.

Les variables prédictives étant toutes binaires, définies sur la même échelle donc, nous pouvons comparer directement les coefficients pour identifier les variables qui contribuent le plus dans le modèle. Nous réalisons un petit histogramme pour avoir une idée de leur distribution.

```
#histogramme des coefficients
hist(coefs)
```

Quelques coefficients seulement, portant des valeurs élevées en valeur absolue, pèsent fortement dans le modèle.



Nous nous intéressons aux 40 premiers.

```
#40 coefficients les plus élevés en valeur absolue
print(sort(abs(coefs),decreasing=T)[1:40])
```

```
       SRI        LEK        GSI        ETG        IRN
0.03661514 0.03604311 0.03464464 0.03457880 0.03449379
       RNT        YTD        IEI        ALE        CMA
0.03430169 0.03382350 0.03373642 0.03366952 0.03352578
       GYC        YLR        VND        GSD        VKD
0.03345874 0.03278617 0.03276628 0.03268657 0.03263532
       PDV        YDP        LGA        FED        ENA
0.03226843 0.03224210 0.03220829 0.03218395 0.03212987
       TYS        EVL        TRH        DHT        STV
0.03201813 0.03197981 0.03189993 0.03189750 0.03183702
       QCD        DLS        SYN        AEG        ADS
0.03175038 0.03151849 0.03148559 0.03141065 0.03137974
       LQP        PTA        AKG        ECD        YSM
0.03135984 0.03135525 0.03132635 0.03122980 0.03119452
       VQD        EHL        PPS        AIE        QVG
0.03119289 0.03119240 0.03112604 0.03103249 0.03099642
```

C'est à ce stade que rentre en scène l'expert métier. Lui seul peut nous éclairer sur l'intérêt de ces combinaisons d'acides aminés.

Evaluation du modèle. Nous utilisons l'échantillon test pour évaluer les performances de notre modèle. Nous calculons les prédictions $\hat{y}_i$ à l'aide de **predict()**.

```
#prédiction sur l'échantillon test
ycp <- mgdkp %>% predict(as.matrix(DProt[-idTrain,-ncol(DProt)]))

#caractéristiques des prédictions
print(summary(ycp))
```

```
       V1
 Min.   :-0.85426
 1st Qu.: 0.02551
 Median : 0.29780
 Mean   : 0.34975
 3rd Qu.: 0.70171
 Max.   : 1.55594
```

Il s'agit de prédictions d'une régression. Elles varient entre min : -0.85246 et max : 1.55594. Comment les convertir en valeurs 1/0 compatibles avec notre problème de classement ?

En cas de codage 0/1 de la variable cible y, on montre (RAK, 2014 ; page 7) que le seuil d'affectation applicable est la moyenne de y, calculé sur l'échantillon d'apprentissage $(\bar{y}_A)$ (aucun paramètre ne doit être calculé sur l'échantillon test).

```
#valeur de référence pour la prédiction
ref_mean <- mean(DProt$y[idTrain])
print(ref_mean)
```

Elle est égale à 0.5490196. Notre règle de classement devient :

$$Si\ \hat{y}_i > \bar{y}_A\ Alors\ \hat{c}_i = 1\ Sinon\ \hat{c}_i = 0$$

```
#classement
cp <- ifelse(ycp > ref_mean,1,0)
print(table(cp))
```

```
0  1
35 15
```

15 individus sont affectés à la famille de protéines 1, 35 à la famille 0.

Confrontons les classes prédites avec les classes observées dans une matrice de confusion.

```
#matrice de confusion
mc <- table(DProt$y[-idTrain],cp)
print(mc)
```

8 = (6 + 2) observations sont mal classées sur les 50 observations constituant l'échantillon test.

```
   cp
     0  1
  0 29  2
  1  6 13
```

Soit, un taux d'erreur de **16%**.

```
#taux d'erreur
err <- 1-sum(diag(mc))/sum(mc)
print(err)
```

Bon, on peut toujours essayer d'améliorer ce taux d'erreur, en passant par une plus forte régularisation par exemple. Mais ce n'est pas le propos de ce tutoriel. La grande nouvelle ici est qu'il est possible de procéder à une régression sur des données comportant 51 observations et 7143 variables explicatives en passant par la descente de gradient.

4 Conclusion

La descente de gradient est une technique d'optimisation utilisable dans le cadre du machine learning. Appliquée à la régression, nous montrons dans ce tutoriel qu'elle permet de retrouver les résultats des implémentations usuelles de la régression linéaire multiple sur des bases appréhendables par ces dernières ; elle est la seule viable en revanche lorsqu'il faut traiter des grandes bases avec une forte dimensionnalité.

5 Références

- (RAK, 2014) Rakotomalala R., « Régression linéaire pour le classement », Avril 2014 ; http://tutoriels-data-mining.blogspot.fr/2014/04/regression-lineaire-pour-le-classement.html
- (RAK, 2018) Rakotomalala R., « Descente de gradient – Diapos », Avril 2018 ; http://tutoriels-data-mining.blogspot.fr/2018/04/descente-de-gradient-diapos.html
- Tutoriel Tanagra, « Régression PLS – Sélection du nombre d'axes », Avril 2008.