KMeans

Anas El Benna et Luana Timofte

5 mars 2019

Introduction

Et voici un nouveau challenge qui a mis en marche nos neurones : le super extra ultra célèbre **K-means**.

Expliquons rapidement ce que c'est ce **K-means**!

Alors, (en essayant de ne pas faire copier coller depuis le site du prof.... oui oui, chut!!!) le K-means c'est un algorithme de classification supervisée. Celui-ci regroupe des observations au sein d'une même urne, selon la distance euclidienne usuelle. En bref : cela nous permet d'identifier des *trucs* automatiquement (intelligence artificielle et tout ça, trop coooool!!!).

L'algorithme

Alors, la logique de l'algorithme (que nous avons appelé **mykmeans**) est plutôt simple. Voici les pas :

1) On a fait une initialisation des centroides (c'est à dire la moyenne des observations contenues dans une urne).

2) On a calculé la distance euclidienne entre chaque observation aux différents centroides.

3) On a affecté à chaque observation l'urne correspondant au centroide le plus proche.

4) On met à jour les centroides.

5) On revient à l'étape *1* jusqu'à convergence.

6) On renvoie la répartition des urnes.

Bon, assez de bavardage ! Voici le code:

```
mykmeans<-function(nbr.classe,data){

  #Initialisation des centroides(1)
  centroides <- matrix(NA,nbr.classe,ncol(data))
  rownames(centroides) <- paste("centre",1:nbr.classe,sep = "_")
  colnames(centroides) <- colnames(data)

  centroides <- as.matrix(data[sample(nrow(data),nbr.classe),])

  #Calcul des distances euclidiennes(2)
  dist.eucl <- function(centroides){
    Dist <- matrix( NA , nrow = nrow(data) , ncol = nbr.classe)

    for (i in 1:nrow(data))
      for (j in 1:nbr.classe)
        Dist[i,j] <- sqrt(sum((data[i,] - centroides[j,])^2))

    return(Dist)
  }
```

```
  while (TRUE){
    #Classiffication(3)
    clusters <- apply(dist.eucl(centroides), 1, which.min)

    test.centroides <- centroides

    #Mise a jour centroides(4)
    for (i in 1:nrow(centroides))
      centroides[i,] <- colMeans(data[clusters==i,])

    if (sum((test.centroides - centroides)^2) < 10^-6)
      break
  }

  #Comptage des chiffres
  Partition <- table(clusters)
  rownames(Partition) <- c(paste(0:9))

  return(list(clusters = clusters, partition = Partition))
}
```

Il est beau, n'est-ce-pas? Maintenant, laissons la beauté de côté, regardons son efficacité !

Application sur MNIST

Le but ici, expliqué d'une façon simple, est de faire deviner des chiffres à notre ordi, grâce au jeu de données **MNIST**, spécifique à l'apprentissage statistique.

Avant de commencer tout le bazar, let's load them all !

```
load("C:/Users/Luana Paully/Downloads/MNIST.RData")
ls()
```

```
## [1] "mykmeans" "x_test"   "x_train"  "y_test"   "y_train"
```

```
#Dimensions
dim(x_test)
```

```
## [1] 10000   784
```

```
dim(x_train)
```

```
## [1] 60000   784
```

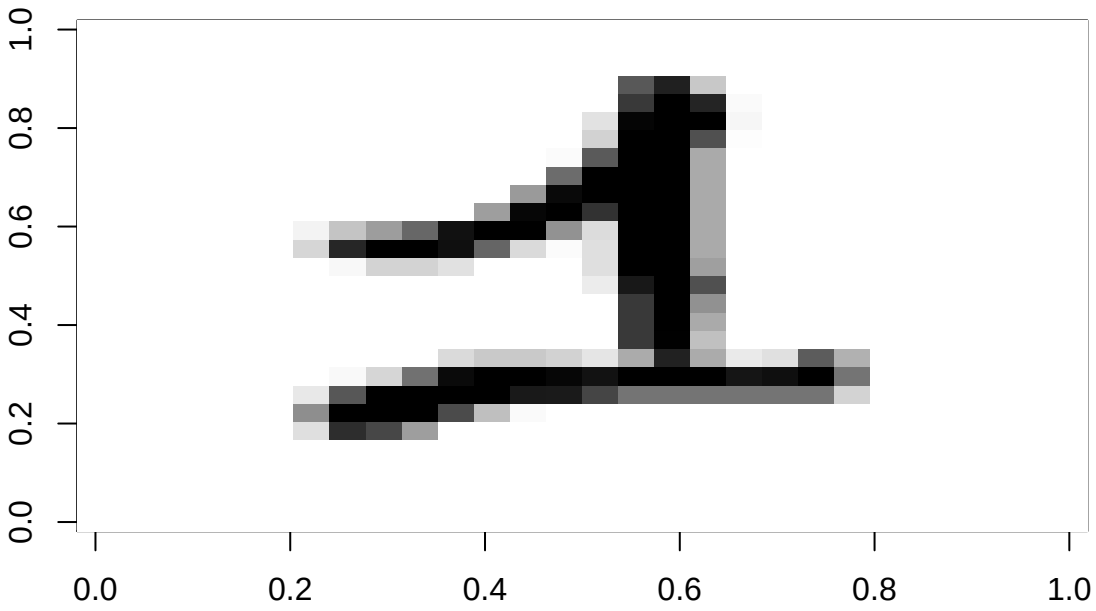
Nous avons les données de **x_train** et **x_test**. Le premier sert à apprendre à notre algorithme les chiffres et le deuxième sert à faire déviner à notre algorithme d'autres chiffres, écrits à la main.

x_train contient 60.000 images pixelisées de dimension 28x28.

x_test contient 10.000 images qui pourront être testées, de même dimension.

Et, pour bien comprendre, voici à quoi tout cela ressemble :

```
image(matrix(x_train[25,],28,28)[,28:1], col=rev(grey(0:255/255)))
```



Et voici le protocole :

1) Classification de K-means.

2) Inspection des classes et calcul de la matrice de confusion.

3) Prédiction sur le jeu de données test.

4) Calcul de la matrice de confusion sur ce dernier jeu.

Syntaxe :

toto$partition - donne le partitionnement de nos images en 10 urnes (chiffres de 0 à 9), avec notre fonction *mykmeans*.

MC_train / MC_test - est la matrice de confusion, qui mesure la qualité du système de classification *(d'après Wikipedia, our saviour)*. Donc on prend le maximum d'une colonne, on regarde son indice qui renvoie à l'urne correspondante (ie: un chiffre entre 0 et 9).

perc - est le pourcentage d'efficacité de notre système de classification.

Commençons d'abord avec *train* :

```
#Classification de Kmeans(1)
toto <- mykmeans(10,x_train)
toto$partition
```

```
## clusters
##     0     1     2     3     4     5     6     7     8     9
##  4713  6352  5327 10345  6749  7645  3093  7332  3028  5416
```

```r
#Matrice de confusion de nos classes de 'train'(2)
MC_train <- table(toto$clusters, as.factor(y_train))
rownames(MC_train) <- c(paste(0:9))

#Pourcentage d'efficacité de mykmeans pour 'train'
perc <- 100*(sum(apply(MC_train,2,max) / colSums(MC_train)) / 10 )

print(MC_train)
```

```
##
##        0    1    2    3    4    5    6    7    8    9
##   0   24   38 4203  241   14   17   78   38   56    4
##   1  163   29  160 1006   14 1457   81   15 3389   38
##   2  192   14  160   59  131  142 4563    2   56    8
##   3   19 6602  684  393  240  871  413  374  575  174
##   4   18    7   58   41 1599  337    1 2777  213 1698
##   5  161   11  333 4022    0 1806   28    3 1190   91
##   6 2499    0   93  112    7  224   98   14   29   17
##   7   19   34   47  178 1593  314    3 2484  176 2484
##   8 2768    0   10   18    2   62  109    7   34   18
##   9   60    7  210   61 2242  191  544  551  133 1417
```

```r
print(perc)
```

```
## [1] 57.35996
```

On enchaine avec *test* :

```r
#Classification de Kmeans(1)
toto <- mykmeans(10,x_test)
toto$partition
```

```
## clusters
##    0    1    2    3    4    5    6    7    8    9
##  876 1134  751  850  719 1200  977 1267 1260  966
```

```r
#Matrice de confustion sur le jeu de données 'test'(2)
MC_test <- table(toto$clusters, as.factor(y_test) )
rownames(MC_test) <- c(paste(0:9))

#Pourcentage d'efficacité de mykmeans pour 'test'
perc <- 100*(sum(apply(MC_test,2,max) / colSums(MC_test)) / 10 )

print(MC_test)
```

```
##
##       0   1   2   3   4   5   6   7   8   9
##   0  20   2  22   5  22  10 787   0   4   4
##   1   1   0  12  11  99  16   1 609  22 363
##   2   1   1 696  17   2   3  13   7   9   2
##   3 786   0  16   2   1   5  21   3   7   9
##   4   1 485 111   2  11  18   7  52  28   4
##   5  58   2  34 351   0 303   5   1 436  10
##   6  71   1  37 521   0 217  37   0  87   6
##   7  39   0  28  27 267 253  25 153 339 136
##   8   3   0  23   8 547  32  12 167  16 452
```

```
##   9   0 644  53  66  33  35  50  36  26  23
```

```
print(perc)
```

```
## [1] 57.65937
```

Comparaison avec KMEANS "normal"

```
real_kmeans=kmeans(x_test,10)
inertie_interne_kmeans <- 100 * (real_kmeans$betweenss / real_kmeans$totss )
inertie_interne_kmeans
```

```
## [1] 26.32421
```

```
#Matrice de confusion du vrai kmeans
MC_kmeans=table(real_kmeans$cluster, y_test)
rownames(MC_kmeans)=c(paste(0:9))
MC_kmeans
```

```
##    y_test
##       0   1   2   3   4   5   6   7   8   9
##   0 814   0  16   4   1   5  21   1   7   8
##   1   5   0 692  49   3   7  17  10  13   2
##   2   1 485 119   5  14  26  10  47  37   6
##   3  55   3  63 703   0 286   2   0 195   7
##   4   0 644  51  62  24  37  51  32  32  19
##   5   2   0   9  13 254  27   2 456  29 450
##   6  60   1  21 150   1 326  44   1 576  13
##   7  23   2  22   6  18  11 784   0   8   2
##   8   3   0  25   8 390  30  24 101  18 289
##   9  17   0  14  10 277 137   3 380  59 213
```

```
accuracy_perc <- 100*(sum(apply(MC_kmeans,2,max) / colSums(MC_kmeans)) / 10 )
accuracy_perc
```

```
## [1] 58.26528
```

On obtient un pourcentage pour notre fonction mykmeans (environ 60%) légerement au dessus de la fonction kmeans de R.

```
$`kmeans`
$`kmeans`$`All.index`
     6      7      8      9     10     11     12     13     14     15
0.0678 0.0731 0.0753 0.0596 0.0590 0.0570 0.0577 0.0618 0.0625 0.0654

$`kmeans`$Best.nc
Number_clusters     Value_Index
         8.0000          0.0753

$`kmeans`$Best.partition
   [1] 8 4 6 3 2 6 8 2 2 8 3 7 8 3 6 4 2 8 4 2 8 7 7 1 2 3 2 2 3 6 4 6 4 7 8 5 8 6 4 6 6 8 2 6 6 4 6 5 2 2 7 4 1 4 5 1 2 6 2 6 2 5 2 5 8 2 7
  [68] 2 4 3 8 3 5 1 6 8 4 8 6 8 2 7 5 2 2 2 8 4 7 6 4 7 6 4 6 2 6 6 7 2 7 3 1 2 2 2 5 6 2 8 1 6 6 2 8 8 2 2 8 8 4 8 2 7 8 2 4 1 1 1 7 7 4 8
 [135] 1 6 4 6 7 8 7 2 4 6 8 6 1 5 3 6 2 2 8 4 6 1 7 7 4 6 2 7 1 2 7 4 2 6 6 2 2 8 4 4 5 6 6 1 6 4 6 4 1 3 1 2 4 4 3 6 6 6 3 2 3 6 6 7 2 5 4
 [202] 7 6 6 6 4 2 4 5 2 2 8 2 6 2 1 4 1 8 1 2 4 5 8 6 5 4 2 6 8 4 6 4 8 8 2 5 5 2 6 4 4 1 2 6 7 3 5 2 6 2 6 2 1 2 2 5 6 5 3 1 6 2 8 2 6 1 6
 [269] 4 3 4 3 6 2 2 2 6 1 5 6 5 8 6 6 2 5 7 2 6 4 2 5 2 5 3 2 3 3 5 1 6 2 6 6 8 3 5 8 8 4 4 3 3 4 6 2 6 6 4 4 6 6 2 6 4 8 5 3 8 6 6 6 6 4 4
 [336] 4 2 2 4 7 6 2 6 4 4 6 1 1 6 2 6 4 4 7 6 1 1 6 2 2 2 7 8 6 2 1 7 4 7 4 2 6 4 1 2 1 2 6 6 2 3 2 4 8 7 6 7 6 6 2 5 1 7 6 8 4 5 4 2 2 5 4
 [403] 4 8 5 2 4 3 4 6 2 6 4 2 2 2 6 8 5 6 8 5 2 5 3 2 8 6 8 2 6 4 2 8 2 8 1 4 8 1 3 3 4 1 1 3 7 8 1 4 4 1 6 4 2 6 6 1 1 3 8 7 7 7 8 1 1 6 5
 [470] 4 1 8 7 6 1 2 6 5 4 4 6 2 2 6 2 3 2 2 8 6 3 1 5 4 8 1 2 6 1 6 4 2 6 5 6 4 6 4 1 4 8 6 6 5 3 6 5 7 1 2 6 2 2 6 4 3 4 4 4 6 2 6 2 2 7 2
 [537] 6 6 6 5 4 2 6 2 2 3 3 5 1 5 8 6 1 7 8 8 2 2 8 7 8 3 2 1 4 8 6 4 7 4 1 2 1 4 4 2 4 2 1 8 1 3 6 5 6 2 3 7 4 2 4 4 3 8 7 4 1 3 2 2 7 6 7
 [604] 1 1 8 1 7 3 6 2 3 5 6 6 6 2 8 4 1 3 1 2 7 6 7 8 2 2 5 2 1 5 5 2 5 2 4 8 8 6 1 3 5 3 1 6 6 4 2 7 2 6 6 4 1 3 4 2 2 2 1 2 6 1 7 8 6 2 4
 [671] 8 2 6 8 4 6 8 4 2 8 7 2 6 4 2 1 4 4 1 8 5 2 8 1 1 6 6 2 2 2 6 3 8 2 3 2 2 8 2 1 1 1 2 3 4 5 6 7 1 2 4 3 2 2 7 6 2 4 4 7 2 5 7 2 4 6 2
 [738] 7 4 1 2 6 3 7 5 6 8 4 2 6 3 4 2 4 6 6 8 2 8 8 8 1 2 3 5 2 4 6 6 7 2 2 6 2 2 5 2 6 4 1 4 4 4 6 2 4 7 4 2 2 6 2 4 4 3 6 5 6 6 5 4 4 2 7
 [805] 3 1 4 4 4 6 8 4 4 8 7 4 4 7 6 4 7 3 2 5 6 2 6 8 5 8 2 6 2 8 6 6 6 4 2 1 6 2 8 2 4 3 8 4 6 4 6 3 2 8 3 4 8 1 5 8 7 7 2 5 1 4 4 4 5 1 1
 [872] 3 1 5 8 5 4 4 4 1 8 2 8 4 4 7 7 4 5 6 4 5 5 2 4 4 3 4 6 4 6 2 2 7 3 5 2 6 8 2 2 8 4 2 1 4 2 2 6 6 6 5 5 4 8 4 5 4 2 6 6 2 3 4 4 8 1 7
 [939] 4 3 7 8 7 6 6 5 2 2 6 6 5 2 7 8 2 4 7 4 4 2 6 1 2 6 6 3 8 6 2 2 4 2 3 7 5 5 4 6 4 6 5 3 4 1 6 5 7 2 6 7 6 4 2 1 6 5 5 3 7 2
 [ reached getOption("max.print") -- omitted 9000 entries ]
```

Figure 1: Meilleurs clusters et nombres de clusters (8 classes) proposés par kmeans

Propositions des meilleurs nombres de clusters de mykmeans, kmeans, CAH_complete et CAH_ward :

```r
library(NbClust)

kmeans_et_CAH <- function( data_images , min_nc=6,max_nc=15, acp =FALSE) {

  res = list()
  indices_vector = c("silhouette")

  if ( acp == TRUE) {
    pca = PCA( data_images , ncp = 2 )
    data_images = pca$ind$coord
  }

  res$mykmeans = NbClust( data=data_images , min.nc = min_nc , max.nc = max_nc ,
                          method = "kmeans" , index = indices_vector )

  res$kmeans = NbClust( data=data_images , min.nc = min_nc , max.nc = max_nc ,
                        method = "kmeans" , index = indices_vector )

  res$cah_complete = NbClust( data=data_images , min.nc = min_nc ,max.nc = max_nc ,
                              method = "complete" , index = indices_vector )

  res$cah_ward = NbClust( data=data_images , min.nc = min_nc , max.nc = max_nc ,
                          method = "ward.D2" , index = indices_vector )

  return(res)
  }
```

```
$cah_complete
$cah_complete$`All.index`
     6      7      8      9     10     11     12     13     14     15
0.0344 0.0367 0.0360 0.0360 0.0341 0.0351 0.0332 0.0338 0.0278 0.0273

$cah_complete$Best.nc
Number_clusters     Value_Index
         7.0000          0.0367

$cah_complete$Best.partition
   [1] 1 2 2 3 4 2 1 1 4 1 5 4 1 6 2 6 2 1 6 2 1 4 2 2 2 7 1 2 3 2 2 2 1 4 1 6 1 2 2 2 2 1 1 2 2 1 2 2 1 1 3 6 2 6 4 3 4 2 1 1 1 3 2 6 2 2 4
  [68] 4 2 4 1 5 3 1 2 2 6 2 2 1 2 4 2 2 2 1 1 1 4 2 6 4 2 6 2 4 2 2 2 2 1 4 5 2 1 1 2 3 2 2 1 2 1 2 1 2 1 2 1 2 4 1 2 6 1 2 2 1 2 5 2 2 2 3 3 6 1
 [135] 2 2 3 2 4 1 3 1 2 2 1 2 2 3 3 1 2 5 2 2 2 2 4 3 6 2 2 3 2 4 4 6 1 1 2 2 2 2 6 6 2 1 2 2 2 2 2 2 2 4 6 1 6 6 7 2 2 2 7 2 5 1 2 3 2 2 6
 [202] 4 2 2 2 2 2 1 2 1 4 1 1 2 2 3 6 1 1 2 2 3 2 1 2 6 2 1 2 1 2 2 2 2 2 2 2 2 2 2 2 1 1 2 2 6 5 1 1 2 1 2 1 1 1 2 2 2 2 5 2 2 1 1 2 2 2 2
 [269] 6 4 1 3 2 1 1 2 2 2 2 2 6 1 2 1 4 2 3 4 2 1 1 6 1 2 6 1 5 4 6 2 1 2 2 1 1 5 2 1 1 6 6 5 3 6 2 1 1 1 6 6 2 1 1 6 3 1 2 7 2 2 2 2 2 2 6
 [336] 6 1 1 5 2 2 2 2 6 2 2 3 2 2 2 2 2 6 4 2 2 2 2 2 1 4 4 2 2 2 3 4 1 3 6 2 6 5 2 2 2 2 2 2 2 5 2 6 2 4 2 3 2 2 2 2 2 4 2 1 6 2 2 4 4 3 2
 [403] 6 2 2 4 6 7 6 2 1 1 6 1 2 2 2 1 2 2 1 3 2 2 5 4 1 2 4 2 2 1 2 3 2 2 2 6 1 3 5 3 1 2 2 4 3 1 2 6 6 6 2 6 1 2 2 2 2 6 1 4 4 4 2 2 2 2 2
 [470] 2 2 1 4 2 2 4 2 2 2 1 2 2 1 3 2 3 1 2 1 2 7 2 6 6 1 6 2 1 2 1 6 1 3 3 2 2 2 2 2 2 2 1 2 2 4 1 2 4 2 1 2 2 1 2 6 5 1 6 2 2 2 2 1 4 3 4
 [537] 2 2 2 2 2 1 2 1 1 3 5 6 2 3 1 2 3 2 1 1 4 1 1 4 1 4 4 2 3 1 1 3 4 1 1 1 2 6 6 2 6 2 2 1 2 5 1 1 2 2 5 4 1 2 6 2 5 1 4 1 2 4 1 1 4 2 4
 [604] 2 2 2 2 4 5 2 1 5 2 2 2 2 2 1 2 2 5 2 2 3 6 4 1 2 1 3 1 2 3 2 1 6 1 6 3 1 2 2 7 2 3 2 2 2 6 2 3 1 2 2 6 2 5 1 1 2 1 2 2 2 2 4 1 1 1 2
 [671] 2 4 2 1 3 2 2 6 1 2 4 1 2 2 2 2 2 6 1 3 1 6 1 1 2 3 2 2 2 4 2 2 7 1 2 7 2 1 1 1 2 2 2 1 5 6 2 2 4 2 4 2 4 1 3 4 2 1 2 2 3 2 3 4 2 2 2 1
 [738] 4 2 3 2 2 7 3 3 2 2 1 1 2 5 6 1 6 2 2 2 2 1 1 1 2 4 5 2 2 6 2 2 3 1 2 2 2 2 2 1 2 2 2 6 1 6 2 1 1 4 1 1 1 2 2 6 2 3 2 2 2 2 2 6 6 1 4
 [805] 5 2 1 6 6 2 2 6 6 1 4 2 2 4 2 6 4 5 1 3 2 2 2 2 6 1 2 2 2 1 3 2 2 2 1 6 2 1 1 4 1 5 1 6 2 6 2 2 1 1 5 6 1 2 2 2 4 4 2 3 2 2 2 2 2 2
 [872] 7 2 2 2 2 6 2 6 2 1 1 1 1 3 2 4 6 2 2 6 6 2 2 6 3 7 2 1 2 2 1 2 4 5 2 2 2 1 1 1 1 6 4 2 2 4 2 2 2 2 2 6 6 2 6 3 6 2 2 1 2 5 6 2 1 2 4
 [939] 1 2 3 1 4 1 2 6 2 2 2 2 1 1 4 1 2 2 2 6 2 2 2 2 1 2 2 3 1 2 2 4 2 1 5 4 2 2 2 2 2 2 2 5 2 2 2 6 3 2 2 4 2 2 2 3 2 3 3 6 7 1 2
 [ reached getOption("max.print") -- omitted 9000 entries ]


$cah_ward
$cah_ward$`All.index`
     6      7      8      9     10     11     12     13     14     15
0.0280 0.0353 0.0402 0.0400 0.0385 0.0389 0.0364 0.0380 0.0398 0.0433

$cah_ward$Best.nc
Number_clusters     Value_Index
        15.0000          0.0433

$cah_ward$Best.partition
   [1]  1  2  3  4  5  3  6  5  7  7  8  9  7  4  3  6  5  1  6  7 10  9  9 11  5  8  1  5  4  3 11  3  6  5 10  2  1  3  2  3  3  1  7  2 11
  [46] 12  3  2  7  5 13  6 12  6  9  4  5  3  7 10  1  2  5 11 10  7  9  5  6  8  1  8  2 10  3  1 11  1 10 10  5  9  2  1 14  5  1  6  9 15
  [91] 11  9 10 11  3  7  3 10  9  5  9  8 12  5  5  5  2 15  7  7 14  1  6  7  1  7  7  5  7  1 12  7  1  9  1  7  8 12 14 12  9  9  6  1 14
 [136]  3  4 15  9  7 13  1  6  3 10  3 14  2  4 10  5  6 10 12  3 12  9  4 11  7  7 13 12  5  9  6  7 10  3  7  7 14  2 11  2  1  3 14  3 14
 [181]  3  6 11  8 14  7  2 12  8  3 15  3  8  5  8 10  3 13  7  2 11  9 15  3  3  6  7  6  2  5  5 10  5 11  7  4 11 10 10 12  1  2  2 10  3
 [226]  2 14  7  3  1 11  6  6 14  1  5  2  2  7  3  6  6 14  1  7  1  8 13  5  2  7  3  7 12  1  1  2  6  2  4 14 10  1  1  7  3 14  3  6  8
 [271]  6  4  3  7  5  5  3 14  2  3  2  7  1 10  5  2 13  5  3  6  6  2  5  2  4  5  8  8  2 14 13  1  3 10  7  8  2  1  7 11  6  8  4  6  3
 [316]  7 13 10  2  6  7  1  5 11  4  7  2  8  1  3  3  2  3 12 11  6  5  1  6  9 11 11  3 11  6  3  4 12  3  6  3 12  8  9 15 14 12  3 10  7
 [361]  5  9  2  2 12  4  9  6  9 11  1  2  8 14 14 12  7  3  3  5  8 11  6  1  9  3  9  2 15  7  2 14  9  3  7  6  2  6  5  5  2 14  6 14 10
 [406]  5  6  8 11  3  1  1  6  1  7  1  3  7  2  3  7  2  7  2  8  7  7  3 10 14  3  6  7 10  7 14 14  6  1 13  8  4  6 10 14  8  9 10 14  6
 [451] 11  6  2 11  5  3  2 13 14  4 12  9  9  9 10 14 14  2 10 12 14  7  9  3 14  5  3  2  9  6  3  7  1 10  7  4 14 10  1  3  8 12  2 11  1
 [496]  4  7  7  4 13  6  5 10  2  3  6  3 12 13 12 10 10  2  2  9 10  2  9 12  7 14  7  5 15  6  8  6 14  6  3  7 11  7  5  9  5  2  3 14  2
 [541] 12  5 14  6  7  8  8  2 11  2  1  7  4 14  1  7  5  1  7  9  7  8  5 14  4  7 13  4  9  6 12  7 14 11 11  7 11  6 14  1 14  8 10  2  3
 [586]  1  8  9  6  7 11 11  8 10  9  6 14  8  7  5  9 14  9 14 12 10 12  9  8  2 13  4  2 10  6 15  7 10 12  3  8 14  5  9  2  9  1  7  5  2
 [631]  5 14  2  2  5  2  1  6 10  7  3 12  8  2  4 12 13  3  6  1 13  1  3  2  6 14  8  6  1  1  7  4  7  3 14  9  1  1  7  6 10  5  3  7 12
 [676]  3 10  6  5  1  9  5  3 11  2 14  6  6 13  7  2  6 12 14 10  3  3  5  1  6  3  8  1  1  8  1  5  7  7 14 12 12 10  8  6  2  3  8 14  5
 [721]  6  8  7  4  9  3 11  6  2  9  5  2  9  5 11  3  5  9  2 11  7  2  8 13  2 15  1  6 10  3  8 12  7  6  3  3  1  7  7  7  7 14  5  8  2
```

Figure 2: Meilleurs clusters et nombres de clusters proposés par CAH_complete (7 classes) et CAH_ward (15 classes)

```
$`mykmeans`
$`mykmeans`$`All.index`
     6      7      8      9     10     11     12     13     14     15
0.0678 0.0731 0.0753 0.0596 0.0590 0.0570 0.0577 0.0618 0.0625 0.0654

$`mykmeans`$Best.nc
Number_clusters     Value_Index
         8.0000          0.0753

$`mykmeans`$Best.partition
   [1] 8 4 6 3 2 6 8 2 2 8 3 7 8 3 6 4 2 8 4 2 8 7 7 1 2 3 2 2 3 6 4 6 4 7 8 5 8 6 4 6 6 8 2 6 6 4 6 5 2 2 7 4 1 4 5 1 2 6 2 6 2 5 2 5 8 2 7
  [68] 2 4 3 8 3 5 1 6 8 4 8 6 8 2 7 5 2 2 2 8 4 7 6 4 7 6 4 6 2 6 6 7 2 7 3 1 2 2 2 5 6 2 8 1 6 6 2 8 8 2 2 8 8 4 8 2 7 8 2 4 1 1 1 7 7 4 8
 [135] 1 6 4 6 7 8 7 2 4 6 8 6 1 5 3 6 2 2 8 4 6 1 7 7 4 6 2 7 1 2 7 4 2 6 6 2 2 8 4 4 5 6 6 1 6 4 6 4 1 3 1 2 4 4 3 6 6 6 3 2 3 6 6 7 2 5 4
 [202] 7 6 6 6 4 2 4 5 2 2 8 2 6 2 1 4 1 8 1 2 4 5 8 6 5 4 2 6 8 4 6 4 8 8 2 5 5 2 6 4 4 1 2 6 7 3 5 2 6 2 6 2 1 2 2 5 6 5 3 1 6 2 8 2 6 1 6
 [269] 4 3 4 3 6 2 2 2 6 1 5 6 5 8 6 6 2 5 7 2 6 4 2 5 2 5 3 2 3 3 5 1 6 2 6 6 8 3 5 8 8 4 4 3 3 4 6 2 6 6 4 4 6 6 2 6 4 8 5 3 8 6 6 6 6 4 4
 [336] 4 2 2 4 7 6 2 6 4 4 6 1 1 6 2 6 4 4 7 6 1 1 6 2 2 2 7 8 6 2 1 7 4 7 4 2 6 4 1 2 1 2 6 6 2 3 2 4 8 7 6 7 6 6 2 5 1 7 6 8 4 5 4 2 2 5 4
 [403] 4 8 5 2 4 3 4 6 2 6 4 2 2 2 6 8 5 6 8 5 2 5 3 2 8 6 8 2 6 4 2 8 2 8 1 4 8 1 3 3 4 1 1 3 7 8 1 4 4 1 6 4 2 6 6 1 1 3 8 7 7 7 8 1 1 6 5
 [470] 4 1 8 7 6 1 2 6 5 4 4 6 2 2 6 2 3 2 2 8 6 3 1 5 4 8 1 2 6 1 6 4 2 6 5 6 4 6 4 1 4 8 6 6 5 3 6 5 7 1 2 6 2 2 6 4 3 4 4 4 6 2 6 2 2 7 2
 [537] 6 6 6 5 4 2 6 2 2 3 3 5 1 5 8 6 1 7 8 8 2 2 8 7 8 3 2 1 4 8 6 4 7 4 1 2 1 4 4 2 4 2 1 8 1 3 6 5 6 2 3 7 4 2 4 4 3 8 7 4 1 3 2 2 7 6 7
 [604] 1 1 8 1 7 3 6 2 3 5 6 6 6 2 8 4 1 3 1 2 7 6 7 8 2 2 5 2 1 5 5 2 5 2 4 8 8 6 1 3 5 3 1 6 6 4 2 7 2 6 6 4 1 3 4 2 2 2 1 2 6 1 7 8 6 2 4
 [671] 8 2 6 8 4 6 8 4 2 8 7 2 6 4 2 1 4 4 1 8 5 2 8 1 1 6 6 2 2 2 6 3 8 2 3 2 2 8 2 1 1 1 2 3 4 5 6 7 1 2 4 3 2 2 7 6 2 4 4 7 2 5 7 2 4 6 2
 [738] 7 4 1 2 6 3 7 5 6 8 4 2 6 3 4 2 4 6 6 8 2 8 8 8 1 2 3 5 2 4 6 6 7 2 2 6 2 2 5 2 6 4 1 4 4 4 6 2 4 7 4 2 2 6 2 4 4 3 6 5 6 6 5 4 4 2 7
 [805] 3 1 4 4 4 4 6 8 4 4 8 7 4 4 7 6 4 7 3 2 5 6 2 6 8 5 8 2 6 2 8 6 6 6 4 2 1 6 2 8 2 4 3 8 4 6 4 6 3 2 8 3 4 8 1 5 8 7 7 2 5 1 4 4 4 5 1 1
 [872] 3 1 5 8 5 4 4 4 1 8 2 8 4 4 7 7 4 5 6 4 5 5 2 4 4 3 4 6 4 6 2 2 7 3 5 2 6 8 2 2 8 4 2 1 4 2 2 6 6 6 5 5 4 8 4 5 4 2 6 6 2 3 4 4 8 1 7
 [939] 4 3 7 8 7 6 6 5 2 2 6 6 5 2 7 8 2 4 7 4 4 2 6 1 2 6 6 3 8 6 2 2 4 2 3 7 5 5 4 6 4 6 5 3 4 1 6 5 7 2 6 7 6 4 2 1 6 5 5 3 7 2
 [ reached getOption("max.print") -- omitted 9000 entries ]
```

Figure 3: Meilleurs clusters et nombres de clusters proposés par mykmeans (8 classes)

Réseau de neurones

And it's not over, mesdames et messieurs ! Procédons à un magnifique **réseau de neurones**, qui va prédire nos chiffres et qui va les ranger dans des urnes. Oh, yes !

Commençons humblement avec l'installation de quelques *packages*, comme **keras, caret, nnet**. Ensuite, on crée les variables pour les données *train* et *test*.

```
install.packages("keras")
install.packages("caret")
install.packages("nnet")
install_keras()
install_keras(tensorflow = "gpu")
```

```
library(keras)
```

```
## Warning: package 'keras' was built under R version 3.5.3
```

```
library(caret)
```

```
## Warning: package 'caret' was built under R version 3.5.3

## Loading required package: lattice

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 3.5.2
```

```
library(nnet)
```

```
## Warning: package 'nnet' was built under R version 3.5.3
```

```
mnist <- dataset_mnist()

#Creation des variables pour les donnees train et test:
train_images <- mnist$train$x
train_labels <- mnist$train$y
test_images <- mnist$test$x
test_labels <- mnist$test$y
```

On convertit les tableaux 3D (images, largeur, longueur) en matrice 2D, en restructurant la longueur et largeur en une seule dimension (ie: des images de dimension 28*28 en vecteurs 784). Ensuite, on convertit les niveaux de gris en réels entre 0 et 1.

```
train_images <- array_reshape(train_images, c(nrow(train_images), 28*28) ) / 255

test_images <- array_reshape(test_images, c(nrow(x_test), 28*28) ) / 255
```

Pour continuer, on utilise la fonction **to_categorical** de *Keras* pour convertir les vecteurs d'entiers entre 0 et 9 en matrices de classes binaires.

```
train_labels <- to_categorical(train_labels, 10)
test_labels <- to_categorical(test_labels, 10)
```

Ensuite, on organise les couches en utilisant le modele sequentiel (**keras_model_sequential()**). L'operateur %>% permet d'empiler les couches lineairement.

Quelques explications :

layer_dense : couche dont les unités sont complètement connectées à la couche précèdente.

layer_dropout : unités ignorées lors de la phase d'entrainement, avec probabilité *"rate=0.4"*, afin d'eviter le surapprentissage.

input_shape : donne la forme des données d'entrée : un vecteur de 784 representant une image de niveaux de gris.

la couche finale : vecteur de taille 10 des probabilites de chaque *'chiffre'*.

```
network <- keras_model_sequential() %>%
  layer_dense(units = 256, activation = "relu", input_shape = c(784)) %>%
  layer_dropout(rate = 0.4 ) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")

summary(network)
```

```
## _________________________________________________________________________
## Layer (type)                 Output Shape              Param #
## =========================================================================
## dense_1 (Dense)              (None, 256)               200960
## _________________________________________________________________________
## dropout_1 (Dropout)          (None, 256)               0
## _________________________________________________________________________
## dense_2 (Dense)              (None, 128)               32896
## _________________________________________________________________________
## dropout_2 (Dropout)          (None, 128)               0
## _________________________________________________________________________
## dense_3 (Dense)              (None, 10)                1290
## =========================================================================
## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
## _________________________________________________________________________
```

Maintenant, on va compiler le modele **(network)** avec la fonction de perte, l'optimisateur et les metriques appropriées.

```
network %>% compile(
  optimizer = "rmsprop",
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)
```

On utilise la fonction **fit()** pour entraîner le modèle sur 48000 échantillons, en une période de longueur 30.

Interprétation :

loss : fonction de perte.

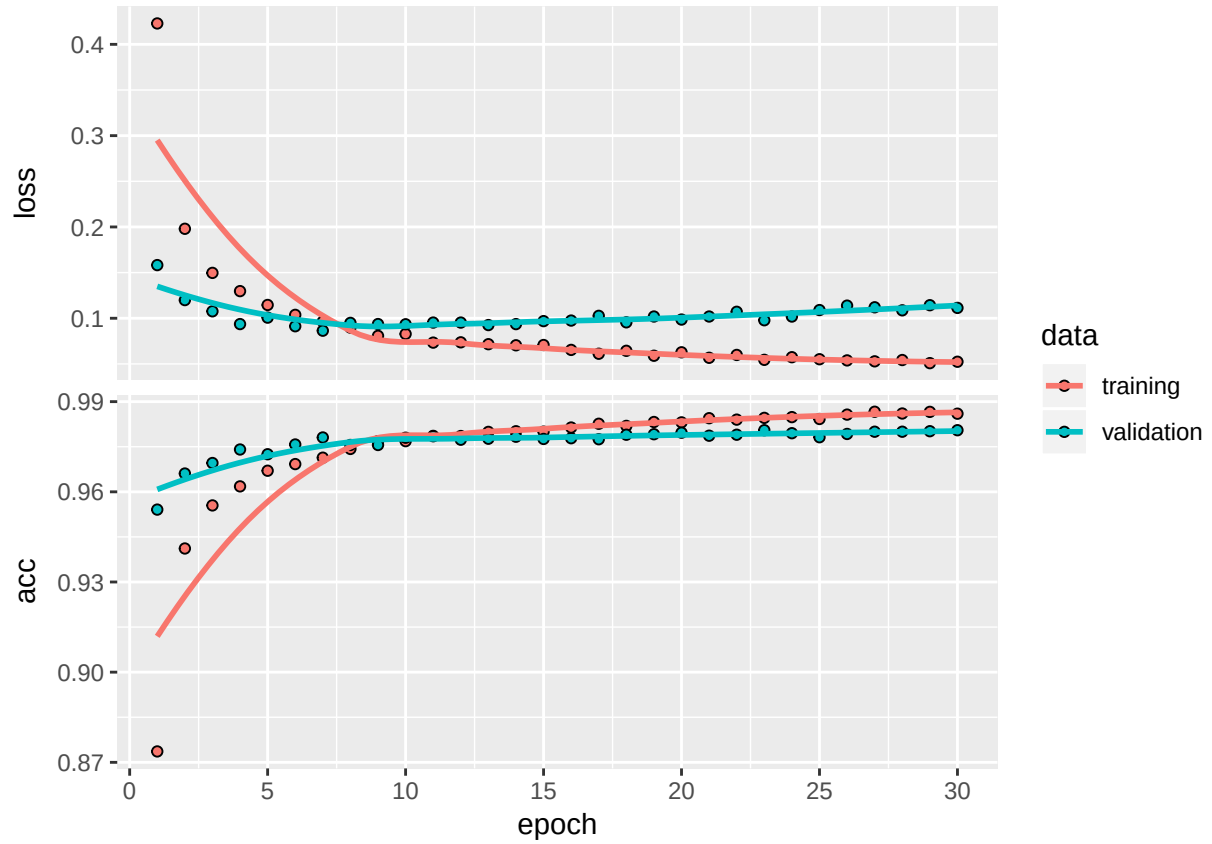
training (en rouge) = Courbes de la perte et précision des données train.

acc = fonction de l'accuracy

validation (en bleu) = Courbes de la perte et précision des données utilisé pour evaluer le modèle.

```
network_training <- network %>% fit(train_images, train_labels, epochs = 30,
                                    batch_size = 128, validation_split = 0.2)
```

```
#Metriques de la perte et la precision
plot(network_training)
```




```
metrics <- network %>% evaluate(test_images, test_labels)
metrics
```

```
## $loss
## [1] 0.100332
##
## $acc
## [1] 0.9816
```

On va générer 100 prédictions (afin d'éviter d'afficher 50 pages de résultats sur le pdf), pour voir si notre algorithme a bien appris à identifier les chiffres.

```
#Genere les predictions sur 'test' (3)
prediction <- network %>% predict_classes(test_images)
prediction[1:100]
```

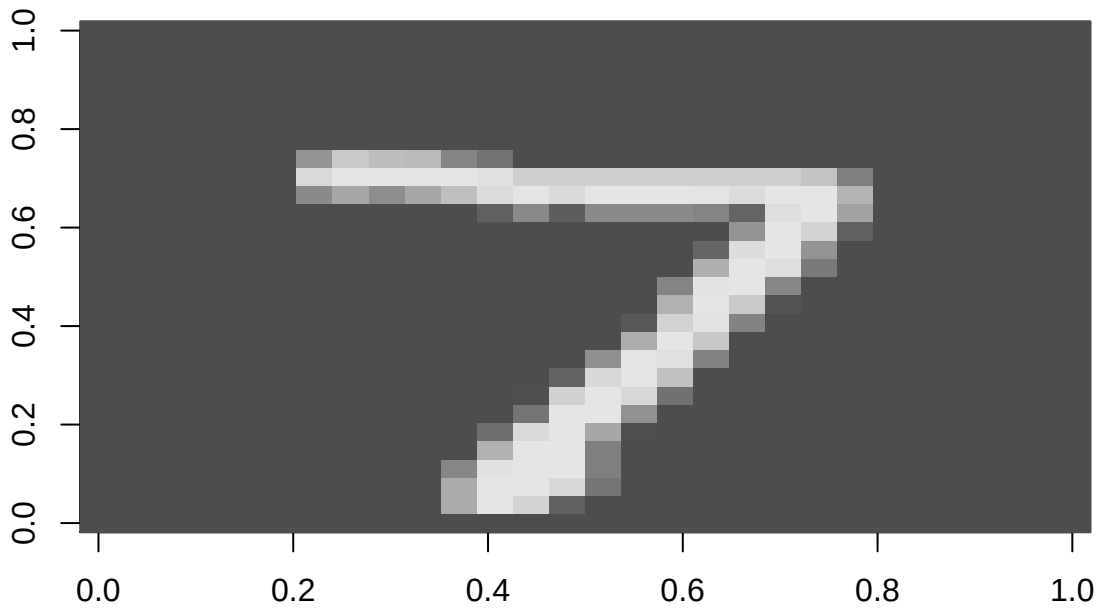
```
##   [1] 7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7
##  [36] 2 7 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0
##  [71] 7 0 2 9 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9
```

Voici une fonction qui montre une image des données test (par exemple la première), à comparer avec la prédiction en-dessus. Ah, mais c'est bien un **7** !

```
affiche_chiffre <- function(X) {
  m <- matrix(unlist(X),nrow = 28,byrow = T)
  m <- t(apply(m, 2, rev))
```

```
  image(m,col=grey.colors(255))
}

#Ligne 1 - colonne 1
affiche_chiffre(test_images[1, ])
```



Et pour s'amuser, on fait une prédiction des classes de 100 images, sur le jeu de données *test* :

Avec un peu de "chance", notre algorithme a tout bien deviné !

```
#100 images pour une meilleure visualisation(3)
par(mfcol = c(10, 10), mar = rep(0, 4))
for (label in 0:9){
  class.idx <- which(prediction == label)

  for (i in sample(class.idx, 10))
    image(matrix(test_images[i,], 28)[,28:1], col = rev(grey(0:255/255)),
          axes = FALSE, bty = "n")
}
```