Université de Technologie de Compiègne
SY09 : Analyse des données et Apprentissage automatique

SY09 : TP 06 : Classifieur euclidien, K plus proches voisin

Julien Jerphanion

Printemps 2018

Table des matières

1 Programmation

Les différentes fonctions développées dans les suites reposent sur des dépendances qu'il s'agit tout d'abord de charger. Cela peut être effectué avec le script suivant :

```
l = list.files("./fonctions/")
for(x in l) {
  source(paste("./fonctions/",x,sep = ""))
}
```

1.1 Classifieur euclidien

Le classifieur euclidien est un des classifieurs les plus simples en apprentissage artificiel. Comme l'algorithme des K-means, il se comprend assez intuitivement : à partir de données représentées dans un espace euclidien fini et dont on connait le classement, on cherche à classer de nouveau point en s'appuyant sur des représentant des classes considérés.

La règle de décision de ce classifieur est de classer un nouveau point selon la classe du représentant des classes le plus proche de celui-ci.

Cette approche de classement intuitive donne lieu à des justifications plus abouties qui sont valables dans un cadre assez plutôt assez souple ; en particulier, son utilisation suppose entre autre : - que les clusters sont représentés de manière égalitaire en termes d'individus ; - que les distribution sous-jacentes des données sont des distributions normales sphériques.

Implémentons maintenant deux fonctions `ceuc.app` et `ceuc.val` pour l'apprentissage d'un tel modèle et pour son classement.

```
ceuc.app <- function(Xapp, zapp) {
  #'
  #' Apprend un classifieur euclidien à partir de données Xapp et d'un
  #' vecteur d'étiquette zapp
  #'
  #' @param Xapp : tableau individu-variables (napp × p)
  #' @param xapp : tableau des étiquettes des individus (longueur napp)
    Xapp <- as.matrix(Xapp)
    zapp <- as.numeric(zapp)

    p <- dim(Xapp)[2]
    g <- max(zapp)

    # Calcul des représentants des classes :
    # ce sont leurs centroïdes
    mu <- matrix(0, nrow=g, ncol=p)
    for (k in 1:g) {
    Xapp_k = Xapp[zapp == k,]
      mu[k,] = apply(X = Xapp_k, MARGIN = 2, mean)
    }

    mu
}
```

```
ceuc.val <- function(mu, Xtst) {
  #'
  #' Réalise un classement d'un tableau individus-variables Xtst.
  #'
```

```
  #' @param mu : paramètres du classifieur euclidien appris
  #' @param Xtst : tableau individu-variables à classer (ntst × p)
    Xtst <- as.matrix(Xtst)

    dist2_X_to_mu = distXY(Xtst ,mu)

    # Règle de décision : prendre la classe dont le représentant est le
    # plus proche du point considéré
    z = apply(X = dist2_X_to_mu, FUN=which.min, MARGIN = 1)

    z
}
```

1.2 K plus proches voisins

Il existe un autre algorithme non paramétriques relativement simple en apprentissage artificiel : l'algorithme des K plus proches voisins. L'idée ici, n'est pas d'apprendre des paramètres en fonction des données mais de se baser directement sur les données pour classer de nouvelles données.

Pour un nouveau point non classifié, on va s'appuyer sur un nombre k de points déjà classifiés pour déduire l'étiquette de celui-ci. On assignera de manière plus précise, ce nouveau point à la classe la plus représentée parmi les classes des k points classifiés.

Pour implémenter cet algorithme, deux fonctions ont été développées ; `kppv.val` et `kppv.tune`.

La première fonction, détermine à partir d'un ensemble de données `Xapp` catégorisées grâce à un vecteur `zapp` les classes de chaque individu du nouveau jeu de données `Xtst` grâce à `k` autres points de `Xapp`.

```
kppv.val <- function(Xapp, zapp, K, Xtst) {
  #'
  #' Déterminer un classement du jeu de données Xtst à partir d'un jeu de données Xapp
  #' et de ces étiquettes zapp.
  #'
  #' @param Xapp : jeu de données pour l'apprentissage
  #' @param zapp : étiquettes de Xapp
  #' @param K : nombre de plus proches voisins à prendre en compte pour le classement
  #' @param Xtst : jeu de données à classer

    Xapp <- as.matrix(Xapp)
    zapp <- as.numeric(zapp)
    Xtst <- as.matrix(Xtst)

    napp <- dim(Xapp)[1]
    ntst <- dim(Xtst)[1]
    p <- dim(Xapp)[2]
    g <- max(zapp)

    # calcul des distances
    d2 <- distXY(Xtst, Xapp)
    d2sor <- t(apply(d2, 1, sort))

    # distance seuil (distance au Kieme plus proche voisin)
    seuil <- d2sor[,K]

    # plus proches voisins
```

```
    is.ppv <- (d2<=matrix(rep(seuil,napp),nrow=ntst,byrow=F))

    # classes des K plus proches voisins
    cl.ppv <- matrix(rep(zapp,ntst),nrow=ntst,byrow=T)*is.ppv

    scores <- matrix(0,nrow=ntst,ncol=g)
    for (k in 1:g)
    {
        scores[,k] <- apply(cl.ppv==k,1,sum)
    }

    # classement en fonction du centre le plus proche
    zpred <- apply(scores, 1, which.max)

    zpred
}
```

La fonction `kppv.tune` détermine le nombre « optimal » de voisins K opt choisi parmi un vecteur `nppv` de valeurs possibles, c'est-à-dire donnant les meilleurs performances sur un ensemble de validation étiqueté (matrice `Xval` de dimensions `nval × p` et vecteur `zval` de longueur `nval`) ; elle prend donc en entrée : — le tableau individus-variables Xapp et le vecteur zapp des étiquettes associées ; — le tableau individus-variables Xval et le vecteur zval à utiliser pour la validation ; — un ensemble de valeurs nppv correspondant aux différents nombres de voisins à tester.

Elle retourne la valeur `Kopt` choisie dans l'ensemble `nppv` et donnant les meilleurs résultats sur `Xval`.

```
kppv.tune <- function(Xapp, zapp, Xval, zval, nppv) {
  #'
  #' Détermine le nombre « optimal » de voisins K opt choisi parmi un vecteur de valeur.
  #'
  #' @param Xapp : jeu de données pour l'apprentissage
  #' @param zapp : étiquettes de Xapp
  #' @param Xval : jeu de données pour la validation
  #' @param zval : étiquettes de Xval
  #' @param nppv : ensemble de valeurs nppv correspondant aux différents nombres de
  #' voisins à tester.

    # Précision des classements
  taux <- rep(0,length(nppv))
    napp = dim(Xapp)[1]
  ind = 0

  # Calcul des taux des precisions
    for (k in nppv) {
      ind = ind + 1
      zpred = kppv.val(Xapp, zapp, k, Xval)
        taux[ind] = sum(zpred == zval) / napp
    }

  # Le meilleur k est celui avec la meilleur précision
    indiceOpt = which.max(taux)
  kOpt = nppv[indiceOpt]
  kOpt
}
```

1.3 Test des fonctions

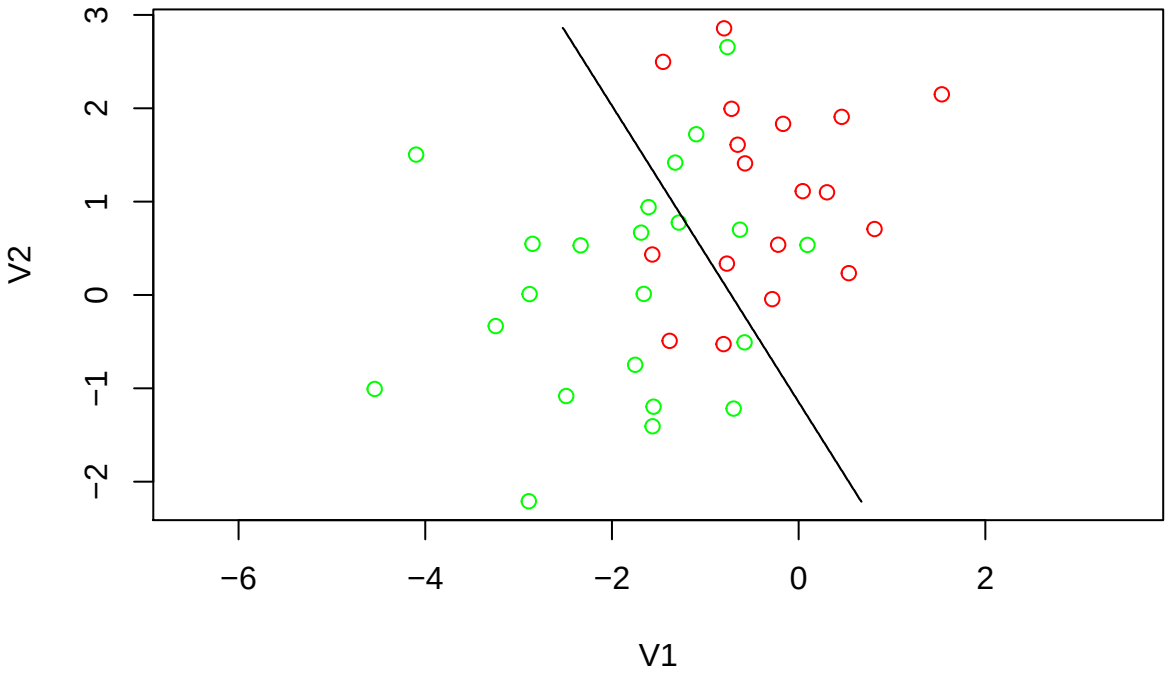
Après avoir implémenté ces deux algorithmes, on peut les tester sur un jeu de données synthétiques.

```
loadData = function(dataSet) {
  #' Charge un jeu de données de deux dimensions

  dataSetName = paste("./donnees/",dataSet,".csv",sep = "")
  donn = read.csv(dataSetName)
  X = donn[,1:2]
  z = donn[,3]

  data = NULL
  data$X = X
  data$z = z

  data
}

data = loadData("Synth1-40")
X = data$X
z = data$z
unique(z)
```

```
## [1] 1 2
```

Ce jeu de donnée comporte ici deux classes et donc deux régions de décisions. Pour vérifier la cohérence des résultats, on peut tracer la frontière de décision entre ces deux régions. Cela se fait avec la fonction `front.ceuc` fournie.

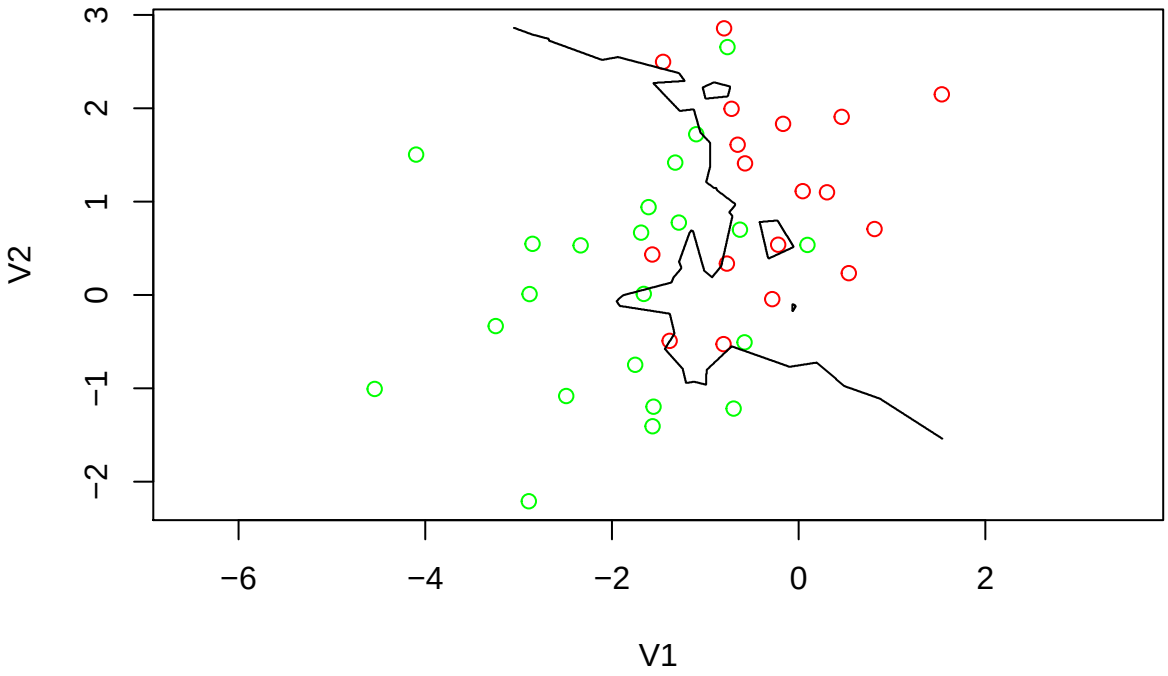
```
# Frontière de décisions pour le classifieur euclidien
mu = ceuc.app(X, z)
front.ceuc(X = X,z = z,mu = mu, discretisation = 1000)
```



On obtient bien une séparation linéaire comme cela a été montré en cours.

On peut aussi tracer la frontière de décision pour l'algorithme des k plus proches voisins grâce à la fonction :

```
# Frontière de décisions pour les k plus proches voisins
front.kppv(X, z, 3, 1000)
```



Cette frontière est obtenue pour $k = 3$. Si on augmente k.

2 Évaluation des performances

Évaluons maintenant les performances de ces algorithmes. Pour cela, on divise généralement le jeu de données original en plusieurs autres jeux: - un jeu d'apprentissage, pour constituer un modèle ; - un jeu de test, pour évaluer les performances de l'algorithme ; - un jeu de validatin, pour paramètrer les meilleurs hyperparamètres pour le modèle après entrainement.

Le jeu de validation n'est pas tout le temps nécessaire, en particulier quand il n'y a pas d'hyper paramètres à déterminer.

On réalise la fonction suivante pour construire les ensembles d'apprentissage et d'entrainement

```
buildTrainTestSets = function(X, z, testSize=0.2, seed=42) {
  #'
  #' Scinde le jeu de données initial et ses étiquettes (X,z) en deux jeux ; $
  #' un d'apprentissage, un de test.
  #'
  #' @param X : jeu de données initial
  #' @param z : étiquettes du jeu de données
  #' @param testSize : proportion du jeu de données
  #' @param seed : graine pour le PRNG
  #'

  # Pour résultats reproductibles
```

```
  set.seed(seed)

  # Paramètres
  napp = dim(X)[1]

  indexTest = napp * (1- testSize)

  # Génération d'une permutation pour le mélange des données
  permutation = sample(1:napp, napp, replace = FALSE)

  # Données permutées
  Xperm = X[permutation,]
  zperm = z[permutation]

  # Créations des ensembles
  Xapp = Xperm[1:indexTest,]
  zapp = zperm[1:indexTest]

  Xtst = Xperm[(indexTest+1):napp,]
  ztst = zperm[(indexTest+1):napp]


  # Objet retourné
  sets = NULL

  sets$Xapp = Xapp
  sets$Xtst = Xtst

  sets$zapp = zapp
  sets$ztst = ztst

  sets
}
```

Construisons donc nos différents jeu pour la suite.

```
sets = buildTrainTestSets(X,z,0.2)

Xapp = sets$Xapp
zapp = sets$zapp

Xtst = sets$Xtst
ztst = sets$ztst
```

2.1 Jeux de données synthétiques

On dispose de cinq jeux de données : `Synth1-40`, `Synth1-100`, `Synth1-500`, `Synth1-1000`, et `Synth2-1000`. Pour chacun de ces jeux de données, chaque classe a été générée suivant une loi normale bivariée. Les distributions sont les mêmes pour les jeux de données `Synth1-40`, `Synth1-100`, `Synth1-500` et `Synth1-1000`, qui ne diffèrent que par le nombre d'exemples disponibles. En revanche, la distribution des données dans `Synth2` est différente.

Estimons les paramètres μ_k et Σ_k des distributions conditionnelles et les proportions de mélanges π_k des classes. Pour cela, on réalise une fonction :

```r
estimateParameters = function(X,z) {
  #'
  #' Éstime les paramètres du jeu de données
  #'
  #' @param X : jeu de données
  #' @param z : étiquette du jeu de données
  #'

  n = dim(X)[1]
  p = dim(X)[2]

  classes = sort(unique(z))
  K = length(classes)

  # Proportion de mélanges:
  pi = c()
  mu = array(dim = c(K, p))
  sigma = array(dim = c(p, p, K))
  for (ind in 1:K) {
    k = classes[ind]
    X_k = X[z == k,]
    pi[ind] = sum(z == k) / n
    mu[ind,] = apply(X=X_k,MARGIN = 2,mean)
    sigma[,,ind] = cov.wt(X_k)$cov
  }

  stopifnot(sum(pi) == 1)

  # Objet de retours
  parameters = NULL

  parameters$classes = classes
  parameters$pi = pi
  parameters$mu = mu
  parameters$sigma = sigma

  parameters
}
```

On peut maintenant estimer les paramètres des jeux de données ainsi :

```r
datasets = c("Synth1-40", "Synth1-100", "Synth1-500", "Synth1-1000")

for(data in datasets) {
  donn = loadData(data)
  X = donn$X
  z = donn$z
  params = estimateParameters(X,z)
  #print(params) # désactivé pour le rendu pdf
}
```

Afin d'estimer les erreurs, on peut effectuer plusieurs classements et calculer les erreurs empiriques.

Pour cela, on réalise deux fonctions qui réalisent cela et donnent aussi les estimations des erreurs pour les deux classifieurs.

```r
# Sur le classifieur euclidien
errorsCeuc = function(X,z, niter=20) {
  #'
  #' Détermination des erreurs pour le classifieur euclidien.
  #'
  #' @param X : jeu de données
  #' @param z : les étiquettes du jeu de données
  #' @param niter : le nombre d'estimation à réaliser

  error = function(z1,z2) {
    #' Calcule l'erreur de classification entre deux
    #' vecteurs d'étiquettes.
    misClassified = 1 * (z1 != z2)
    sum(misClassified) / length(z1)
  }

  errApp = c()
  errTest = c()
  for (i in 1:niter) {
    donn.sep <- separ1(X, z)
    Xapp <- donn.sep$Xapp
    zapp <- donn.sep$zapp
    Xtst <- donn.sep$Xtst
    ztst <- donn.sep$ztst

    mu = ceuc.app(Xapp,zapp)

    zappClass = ceuc.val(mu, Xapp)
    ztstClass = ceuc.val(mu, Xtst)

    errApp[i] = error(zapp,zappClass)
    errTest[i] = error(ztst,ztstClass)
  }

  estErrApp = mean(errApp)
  estErrTest = mean(errTest)

  # Objet de retour
  errors = NULL
  errors$estErrApp = estErrApp
  errors$estErrTest = estErrTest

  errors$errApp = errApp
  errors$errTest = errTest

  errors
}
```

On on cherche le k optimal avec comme ensemble de validation l'ensemble d'apprentissage, on trouve naturellement 1. Cela est normal, car chaque point de l'ensemble de validation est exactement sur un point de l'ensemble d'apprentissage : lui-même !

```r
nnpv = 1 + 2 * c(0,1,2,3,4,5)
kppv.tune(Xapp = X,zapp = z,Xval=X,zval=z,nnpv)
```

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

Il faut pour cela utiliser un ensemble de validation dédié, c'est ce que nous nous proposons de réaliser dans la suite.

```
errorsKppv = function(X, z, niter=20, nnpv=NULL) {
  #' Détermination des erreurs pour le classifieur des k plus proches voisins.
  #'
  #' @param X : jeu de données
  #' @param z : les étiquettes du jeu de données
  #' @param niter : le nombre d'estimation à réaliser
  #' @param nnpv : valeurs à tester pour K

  error = function(z1,z2) {
    #' Calcule l'erreur de classification entre deux
    #' vecteurs d'étiquettes.
    misClassified = 1 * (z1 != z2)
    sum(misClassified) / length(z1)
  }

  errApp = c()
  errTest = c()
  kOpts = c()
  for (i in 1:niter) {
    donn.sep <- separ2(X, z)
    Xapp <- donn.sep$Xapp
    zapp <- donn.sep$zapp
    Xval <- donn.sep$Xval
    zval <- donn.sep$zval
    Xtst <- donn.sep$Xtst
    ztst <- donn.sep$ztst

    if (is.null(nnpv))
      nnpv = 1 + 2 * c(0,1,2,3,4,5,6)

    kOpt = kppv.tune(Xapp,zapp,Xval,zval,nnpv)
    kOpts[i] = kOpt

    zappClass = kppv.val(Xapp, zapp, kOpt, Xapp)
    ztstClass = kppv.val(Xapp, zapp, kOpt, Xtst)

    errApp[i] = error(zapp,zappClass)
    errTest[i] = error(ztst,ztstClass)
  }

  estErrApp = mean(errApp)
  estErrTest = mean(errTest)
  estkOpt = mean(kOpts)

  # Objet de retour
  errors = NULL
  errors$estErrApp = estErrApp
  errors$estErrTest = estErrTest
  errors$estkOpt = estkOpt
```

```
  errors$errApp = errApp
  errors$errTest = errTest
  errors$kOpts = kOpts

  errors
}
```

Estimons les erreurs sur les différents jeux de données :

```
for(data in datasets) {
  donn = loadData(data)
  X = donn$X
  z = donn$z
  print(paste(data,"Ceuc","kppv",sep = "|"))
  errCeuc = errorsCeuc(X,z)
  errKppv = errorsKppv(X,z)
  print(paste(" - Erreur Xapp", errCeuc$estErrApp,errKppv$estErrApp,sep = "|"))
  print(paste(" - Erreur Xtst", errCeuc$estErrTest,errKppv$estErrTest,sep= "|"))
}
```

```
## [1] "Synth1-40|Ceuc|kppv"
## [1] " - Erreur Xapp|0.216666666666667|0.15"
## [1] " - Erreur Xtst|0.238461538461538|0.29"
## [1] "Synth1-100|Ceuc|kppv"
## [1] " - Erreur Xapp|0.0955223880597015|0.037"
## [1] " - Erreur Xtst|0.0878787878787879|0.104166666666667"
## [1] "Synth1-500|Ceuc|kppv"
## [1] " - Erreur Xapp|0.133333333333333|0.1168"
## [1] " - Erreur Xtst|0.133832335329341|0.1552"
## [1] "Synth1-1000|Ceuc|kppv"
## [1] " - Erreur Xapp|0.138605697151424|0.1296"
## [1] " - Erreur Xtst|0.150150150150150|0.1572"
```

L'erreur de test est plus grande que l'erreur d'entrainement car l'algorithme a appris à partir des données d'entrainement et sait donc mieux reconnaître ces données. Il semblerait que le nombre de voisins à considérer pour le classement augmentent avec le nombre d'individus dans le jeu de données.

Il ne semble pas y avoir un algorithme qui ait de meilleurs résultats dans le cadre général.

Intéressons nous au jeu `Synth2-1000` maintenant

```
data = "./donnees/Synth2-1000.csv"
donn = read.csv(data)
X = donn[,1:2]
z = donn[,3]
estimateParameters(X,z)
```

```
## $classes
## [1] 1 2
##
## $pi
## [1] 0.523 0.477
##
## $mu
##           [,1]         [,2]
## [1,]  3.018388 -0.006382269
## [2,] -2.142281 -0.026524483
```

```
##
## $sigma
## , , 1
##
##           [,1]      [,2]
## [1,] 0.9904026 0.1131386
## [2,] 0.1131386 1.0928705
##
## , , 2
##
##            [,1]       [,2]
## [1,]  4.4347816 -0.1543981
## [2,] -0.1543981  1.0308554
```

```
errCeuc = errorsCeuc(X,z)
errKppv = errorsKppv(X,z)

print(paste(data,"Ceuc","kppv",sep = "|"))
```

```
## [1] "./donnees/Synth2-1000.csv|Ceuc|kppv"
```

```
print(paste(" - Erreur Xapp", errCeuc$estErrApp,errKppv$estErrApp, sep = "|"))
```

```
## [1] " - Erreur Xapp|0.0620689655172414|0.0479"
```

```
print(paste(" - Erreur Xtst", errCeuc$estErrTest,errKppv$estErrTest, sep = "|"))
```

```
## [1] " - Erreur Xtst|0.0629129129129129|0.06"
```

On peut s'intéresser au intervalles de confiance pour les erreurs.

Voici ceux qui se rattachent au classifieur euclidien.

```
t.test(errCeuc$errApp,conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errApp
## t = 71.669, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.06025631 0.06388162
## sample estimates:
##  mean of x
## 0.06206897
```

```
t.test(errCeuc$errTest,conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errTest
## t = 27.727, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.05816374 0.06766209
## sample estimates:
##  mean of x
```

```
## 0.06291291
```

Et voici ceux qui se rattachent aux k plus proches voisins.

```
t.test(errKppv$errApp,conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errApp
## t = 15.076, df = 19, p-value = 5.033e-12
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.04125017 0.05454983
## sample estimates:
## mean of x
##    0.0479
```

```
t.test(errKppv$errTest,conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errTest
## t = 18.875, df = 19, p-value = 9.102e-14
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.05334653 0.06665347
## sample estimates:
## mean of x
##      0.06
```

2.2 Jeux de données réelles

Appliquons cela à deux jeux de données réels ; les jeux de données `Pima` et `BreastCancer`.

```
donn = read.csv("./donnees/Pima.csv")
X = donn[,1:7]
z = donn[,8]

estimateParameters(X,z)
```

```
## $classes
## [1] 1 2
##
## $pi
## [1] 0.6672932 0.3327068
##
## $mu
##          [,1]     [,2]     [,3]     [,4]     [,5]      [,6]     [,7]
## [1,] 2.926761 110.0169 69.91268 27.29014 31.42958 0.4463155 29.22254
## [2,] 4.700565 143.1186 74.70056 32.97740 35.81977 0.6165876 36.41243
##
## $sigma
## , , 1
##
```

```
##              [,1]        [,2]        [,3]         [,4]         [,5]
## [1,]  7.768632132   4.6198854   6.6320045   3.66537758  0.008387045
## [2,]  4.619885414 589.8528209  56.2528925  32.56852869 25.477747275
## [3,]  6.632004456  56.2528925 141.6844434  28.70337392 23.084793507
## [4,]  3.665377576  32.5685287  28.7033739 101.61332060 44.318512772
## [5,]  0.008387045  25.4777473  23.0847935  44.31851277 42.860958861
## [6,] -0.041084173   0.6602771  -0.1322944   0.05539973  0.094909004
## [7,] 18.304479987  42.9114825  39.4234423  17.46349964  4.470518023
##              [,6]        [,7]
## [1,] -0.04108417 18.30447999
## [2,]  0.66027714 42.91148245
## [3,] -0.13229441 39.42344235
## [4,]  0.05539973 17.46349964
## [5,]  0.09490900  4.47051802
## [6,]  0.08926997  0.06475163
## [7,]  0.06475163 98.07745683
##
## , , 2
##
##              [,1]        [,2]        [,3]        [,4]        [,5]
## [1,] 15.35869286  -9.8733629   6.1371020  -4.1431690  -4.6531362
## [2,] -9.87336287 977.5028891  32.8425462  31.1788328  10.2419588
## [3,]  6.13710195  32.8425462 156.8473292  12.3568310  18.0127729
## [4,] -4.14316898  31.1788328  12.3568310 108.0563046  35.5538585
## [5,] -4.65313624  10.2419588  18.0127729  35.5538585  43.7127318
## [6,] -0.09445374   0.2328163  -0.1780276   0.5365418   0.3883173
## [7,] 23.52760657  34.6894260  36.2719248  -7.4394902 -13.7672926
##              [,6]        [,7]
## [1,] -0.09445374  23.5276066
## [2,]  0.23281626  34.6894260
## [3,] -0.17802761  36.2719248
## [4,]  0.53654176  -7.4394902
## [5,]  0.38831729 -13.7672926
## [6,]  0.15914902  -0.1502267
## [7,] -0.15022666 117.4482537
```

```
errCeuc = errorsCeuc(X,z)
errKppv = errorsKppv(X,z)

print(paste(data,"Ceuc","kppv",sep = "|"))
```

```
## [1] "./donnees/Synth2-1000.csv|Ceuc|kppv"
```

```
print(paste(" - Erreur Xapp", errCeuc$estErrApp,errKppv$estErrApp, sep = "|"))
```

```
## [1] " - Erreur Xapp|0.243521126760563|0.191917293233083"
```

```
print(paste(" - Erreur Xtst", errCeuc$estErrTest,errKppv$estErrTest, sep = "|"))
```

```
## [1] " - Erreur Xtst|0.254519774011299|0.258646616541353"
```

```
t.test(errCeuc$errApp, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errApp
```

```
## t = 61.744, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.2352661 0.2517761
## sample estimates:
## mean of x
## 0.2435211
```

```
t.test(errCeuc$errTest, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errTest
## t = 45.596, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.2428364 0.2662031
## sample estimates:
## mean of x
## 0.2545198
```

```
t.test(errKppv$errApp, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errApp
## t = 33.191, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.1798150 0.2040196
## sample estimates:
## mean of x
## 0.1919173
```

```
t.test(errKppv$errTest, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errTest
## t = 35.109, df = 19, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.2432275 0.2740657
## sample estimates:
## mean of x
## 0.2586466
```

```
donn = read.csv("donnees/Breastcancer.csv")
X = donn[,1:9]
z = donn[,10]

estimateParameters(X,z)
```

```
## $classes
```

```
## [1] 1 2
##
## $pi
## [1] 0.6500732 0.3499268
##
## $mu
##          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
## [1,] 2.963964 1.306306 1.414414 1.346847 2.108108 2.414414 2.083333
## [2,] 7.188285 6.577406 6.560669 5.585774 5.326360 4.707113 5.974895
##          [,8]     [,9]
## [1,] 1.261261 1.065315
## [2,] 5.857741 2.602510
##
## $sigma
## , , 1
##
##              [,1]      [,2]          [,3]       [,4]         [,5]
##  [1,]  2.79779554 0.3948102  4.777215e-01 0.39175971  0.231895552
##  [2,]  0.39481016 0.7321498  5.702926e-01 0.22083257  0.307668843
##  [3,]  0.47772151 0.5702926  9.159091e-01 0.21033901  0.289183088
##  [4,]  0.39175971 0.2208326  2.103390e-01 0.84105098  0.235556098
##  [5,]  0.23189555 0.3076688  2.891831e-01 0.23555610  0.769324629
##  [6,]  0.21361316 0.4822565  4.125231e-01 0.35029386  0.298212434
##  [7,]  0.17908202 0.2407825  1.978932e-01 0.11324304  0.142212190
##  [8,]  0.32772050 0.3983487  3.564965e-01 0.22294755  0.366725642
##  [9,] -0.03375836 0.0205804 -4.067273e-05 0.02921319 -0.007077054
##           [,6]        [,7]       [,8]          [,9]
##  [1,] 0.2136132  0.17908202 0.32772050 -3.375836e-02
##  [2,] 0.4822565  0.24078254 0.39834869  2.058040e-02
##  [3,] 0.4125231  0.19789315 0.35649645 -4.067273e-05
##  [4,] 0.3502939  0.11324304 0.22294755  2.921319e-02
##  [5,] 0.2982124  0.14221219 0.36672564 -7.077054e-03
##  [6,] 1.4892726  0.27915726 0.34069510  5.413540e-02
##  [7,] 0.2791573  1.12848006 0.34838224 -2.351392e-02
##  [8,] 0.3406951  0.34838224 0.91127245  2.804385e-02
##  [9,] 0.0541354 -0.02351392 0.02804385  2.598326e-01
##
## , , 2
##
##             [,1]       [,2]       [,3]      [,4]       [,5]        [,6]
##  [1,]  5.9433916  0.6471291  0.7049154 -1.123361  0.1021589 -0.42361380
##  [2,]  0.6471291  7.4215042  5.0446538  2.790602  3.0670687 -0.41000668
##  [3,]  0.7049154  5.0446538  6.6002953  2.195405  2.4044865 -0.24686192
##  [4,] -1.1233606  2.7906016  2.1954045 10.218452  1.5055026 -1.73947822
##  [5,]  0.1021589  3.0670687  2.4044865  1.505503  5.9686720 -0.24854963
##  [6,] -0.4236138 -0.4100067 -0.2468619 -1.739478 -0.2485496  7.05671390
##  [7,] -0.1002953  2.4179178  1.9847228  2.464347  1.2057066 -0.61242572
##  [8,] -0.1075560  2.7295454  2.6641293  1.978640  1.8911606  0.34471362
##  [9,]  0.7390211  1.6842586  1.3834605  1.649784  2.0882529  0.02594845
##             [,7]       [,8]       [,9]
##  [1,] -0.1002953 -0.1075560 0.73902113
##  [2,]  2.4179178  2.7295454 1.68425864
##  [3,]  1.9847228  2.6641293 1.38346050
##  [4,]  2.4643472  1.9786400 1.64978376
```

```
##  [5,]  1.2057066  1.8911606 2.08825287
##  [6,] -0.6124257  0.3447136 0.02594845
##  [7,]  5.2094511  1.9375901 0.34712211
##  [8,]  1.9375901 11.2149713 1.90960233
##  [9,]  0.3471221  1.9096023 6.57663233
```

```
errCeuc = errorsCeuc(X,z)
errKppv = errorsKppv(X,z)

print(paste(data,"Ceuc","kppv",sep = "|"))
```

```
## [1] "./donnees/Synth2-1000.csv|Ceuc|kppv"
```

```
print(paste(" - Erreur Xapp", errCeuc$estErrApp,errKppv$estErrApp, sep = "|"))
```

```
## [1] " - Erreur Xapp|0.0368131868131868|0.0279239766081871"
```

```
print(paste(" - Erreur Xtst", errCeuc$estErrTest,errKppv$estErrTest, sep = "|"))
```

```
## [1] " - Erreur Xtst|0.0445175438596491|0.0344117647058824"
```

```
t.test(errCeuc$errApp, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errApp
## t = 24.22, df = 19, p-value = 9.562e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.03363186 0.03999451
## sample estimates:
##  mean of x
## 0.03681319
```

```
t.test(errCeuc$errTest, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errCeuc$errTest
## t = 14.486, df = 19, p-value = 1.014e-11
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.03808556 0.05094953
## sample estimates:
##  mean of x
## 0.04451754
```

```
t.test(errKppv$errApp, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errApp
## t = 9.4153, df = 19, p-value = 1.377e-08
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.02171647 0.03413148
```

```
## sample estimates:
##  mean of x
## 0.02792398
```

```
t.test(errKppv$errTest, conf.level = 0.95)
```

```
##
##  One Sample t-test
##
## data:  errKppv$errTest
## t = 10.301, df = 19, p-value = 3.256e-09
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.02741995 0.04140358
## sample estimates:
##  mean of x
## 0.03441176
```

.