Variantes des k-means pour tableaux de distances

Ramón Daniel Regueiro Espiño

```
rm(list=ls())
graphics.off()
```

Un algorithme des k-medoïdes

1. Programmer l'agorithme décrit dans la section 2 de l'article scientifique joint au sujet.

Pour programmer l'algorithme, on va créer trois fonctions auxiliares qui correspondent à chaque pas de l'algorithme. Après, on fera une fonction `algo()` pour appliquer l'algorithme.

On commence pour faire la premier fonction `init()`. Ce fonction est l'équivalent à l'initialisation du algorithme, ce qui se correspond au pas 1 de la composition décrit dans l'article. Dans ce cas, le but est de faire une sélection initiale des medoïds. Comme on peut voir dans le tableaux 7 du article, ce mode de sélection peut être différent, mais les résultats obtenus peuvent être pires avec une autre sélection initiale.

```
init<-function(X,k){
  n<-dim(X)[1]
  p<-dim(X)[2]
  obj.dist<-matrix(0,nrow=n,ncol=n)
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      obj.dist[i,j]<-sqrt(sum((X[i,]-X[j,])^2))
      obj.dist[j,i]<-obj.dist[i,j]
    }
  }
  obj.drow<-rowSums(obj.dist)
  v<-rep(0,n)
  for (j in 1:n){
    som<-0
    for (i in 1:n){
      som<-som+obj.dist[i,j]/obj.drow[i]
    }
    v[j]<-som
  }
  vs<-sort(x=v)
  cluster<-rep(0,n)
  medoids<-rep(0,k)
  for (i in 1:k){
    medoids[i]<-which(v==vs[i])
    cluster[medoids[i]]<-i
  }
  dist2<-obj.dist[,medoids]
  dist<-0
```

```
  for (i in 1:n){
    valmin<-which.min(dist2[i,])
    dist<-dist+dist2[i,valmin] #On l'ajoute ici pour ne créer pas autre boucle for après
    cluster[i]<-cluster[medoids[valmin]]
  }
  return(list("medoids"=medoids,"groupes"=cluster,"dist"=dist,"distances"=obj.dist))
}
```

Dans ce cas, on retournera aussi les medoids obtenus car cet un argument de la prochaine fonction, la distance car on l'utilisera pour la condition d'arrêt et la matrice de distances entre les pairs de données pour éviter de le calculer après une autre fois, ce qui est consideré clé pour la vitesse de l'algorithme dans l'article.

La prochaine fonction à créer est laquelle se correspond au pas 2 du algorithme. Le but de cette fonction `upd()` est l'*update medoids*. Dans ce cas, et pour une vitesse majeure, on pourrait la combiner avec la fonction du pas 3, mais pour une compréhension plus facile on a crée deux fonctions différentiées.

```
upd<-function(cluster,medoids,obj.dist){
  k<-length(medoids)
  for (i in 1:k){
    clus<-which(cluster==i)
    k.n<-length(clus)
    k.dist<-rep(0,k.n)
    for (j in 1:k.n){
      for (l in 1:k.n){
        k.dist[j]<-k.dist[j]+obj.dist[clus[l],clus[j]]
      }
    }
    ind<-which.min(k.dist)
    medoids[i]<-clus[ind]
  }
  return(medoids)
}
```

Pour le pas 3, ce qui a comme but l'assignation de nos données à chaque medoïd, on developpe la fonctionne `asig()` qui retourne aussi la nouvelle distance pour éviter de créer un autre boucle après.

```
asig<-function(cluster,medoids,obj.dist){
  k<-length(medoids)
  n<-dim(obj.dist)[1]
  clus.new<-rep(0,n)
  dist.new<-0
  for (i in 1:n){
    j<-which.min(obj.dist[i,medoids])
    clus.new[i]<-j
    dist.new<-dist.new+obj.dist[i,j]
  }
  return(list("dist"=dist.new,"cluster"=clus.new))
}
```

Pour finaliser cette première partie, on va créer une fonction `algo()` qui contient l'ensemble des autres fonctions. On va introduire aussi un nombre maximum d'itérations pour le cas de non convergence. La fonction retournera les clusters, les medoïds, le nombre d'itérations nécessaire et si se vérifie le critère de convergence.

```
algo<-function(X,k,nit=NULL){
  X<-as.matrix(X)
```

```
  #C'est plus vite de travailler avec une matrice que avec un data frame
  maxit<-10^3
  #On utilise un nombre maximum d'itérations comme condition d'arrêt
  conv<-FALSE
  if(!is.null(nit)){
    maxit<-nit
  }
  val<-init(X,k) #Initialisation
  medoids<-val$medoids
  groupes<-val$groupes
  obj.dist<-val$distances
  dist0<-val$dist
  for (its in 1:maxit){
    medoids<-upd(groupes,medoids,obj.dist)
    prob<-asig(cluster=groupes,medoids=medoids,obj.dist=obj.dist)
    groupes<-prob$cluster
    dist1<-prob$dist
    if (dist1>=dist0){
      conv<-TRUE
      break
    }
    dist0<-dist1
  }
  return(list("conv"=conv,"nit"=its,"medoids"=medoids,"clusters"=groupes))
}
```

2. Simuler des données suivant le protocole décrit par la section 3.3 et le tableau 5 de l'article.

Une fois qu'on a programmé l'algorithme il faut le tester. Alors, on va faire des simulations pour obtenir des données et après tester notre algorithme. On fera des comparations entre les résultats obtenus en utilisant l'algorithme dévelopé et autres algorithmes comme le `sk-means`.

Pour faire des simulations et ne changer pas le résultat obtenu chaque fois, on va fixer une semence initiale.

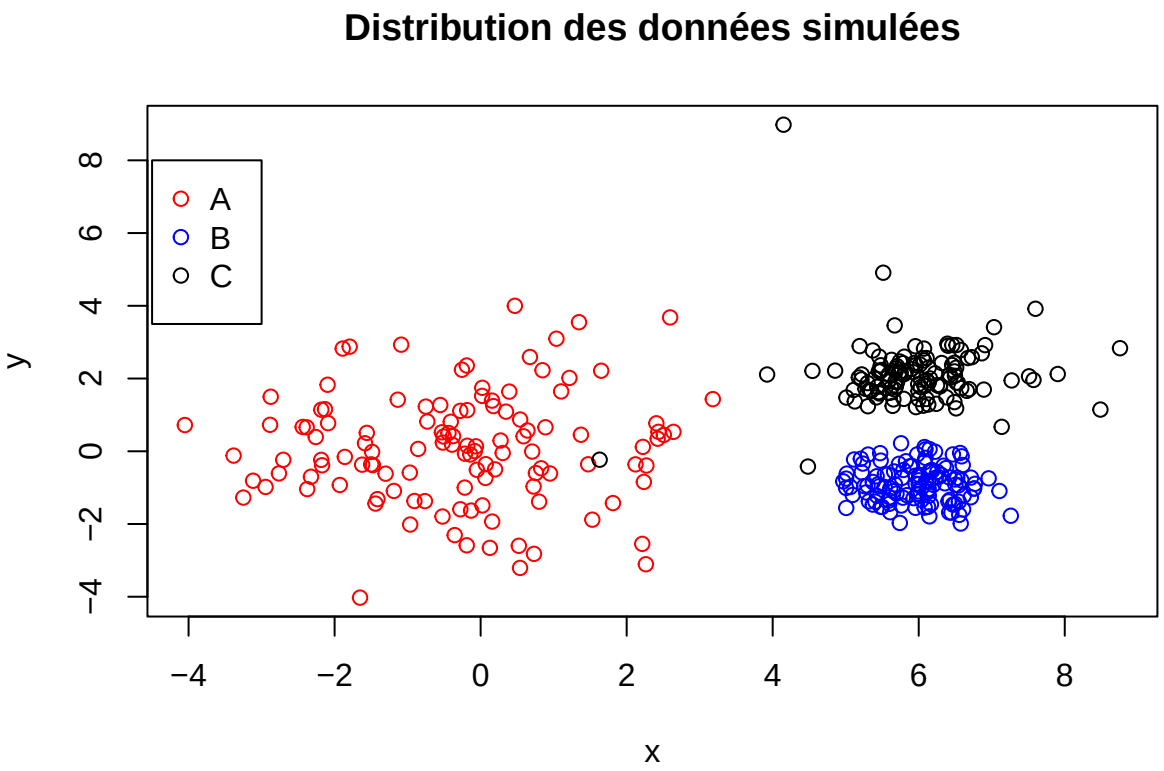
```
set.seed(2021)
```

Pour notre simulation, on fera en premier lieu le groupe A, après le B et finalement le C. Pour chaque groupe, on simulera en premier lieu les valeurs de la variable x et après les valeurs de la variable y. Pour le groupe C, on simulera en avant les données avec la mineur variance et après les autres.

Dans ce cas, pour visualiser un exemple de données simulées avec bruit on utilisera un 10% des données de C avec σ_L^C.

```
n<-120
L<-0.10
A.x<-rnorm(n,mean=0,sd=1.5)
A.y<-rnorm(n,mean=0,sd=1.5)
B.x<-rnorm(n,mean=6,sd=0.5)
B.y<-rnorm(n,mean=-1,sd=0.5)
#12 données du type noise, donc 108 de l'autre type
C.x<-rnorm(n=n-as.integer(n*L),mean=6,sd=0.5)
C.y<-rnorm(n=n-as.integer(n*L),mean=2,sd=0.5)
C.xn<-rnorm(n=as.integer(n*L),mean=6,sd=2)
C.yn<-rnorm(n=as.integer(n*L),mean=2,sd=2)
C.x<-c(C.x,C.xn)
```

```
C.y<-c(C.y,C.yn)
dat.x<-c(A.x,B.x,C.x)
dat.y<-c(A.y,B.y,C.y)
dat.cl<-c(rep("A",120),rep("B",120),rep("C",120))
sim.dat<-data.frame("x"=dat.x,"y"=dat.y,"class"=dat.cl)
plot(x=sim.dat$x,y=sim.dat$y,col=c(rep("red",120),rep("blue",120),rep("black",120)),
     pch=1,xlab="x",ylab="y",main="Distribution des données simulées")
legend(x=c(-4.5,-3),y=c(3.5,8),legend=c("A","B","C"), col=c("red","blue","black"),pch=1)
```



On peut voir comme certains données de C, lesquelles étaient simulés avec σ_L^C, sont moins concentrés que les autres. De plus, si on ne connais pas le groupe a qui fait partie chaque donnée, il semble que au moins la donnée de la class C plus à gauche fait partie de la class A.

3. Comparer votre algorithme avec `PAM` du package `cluster` de `R`, les k-means et `mclust`. Attention, pour une comparaison fiable il faut répéter les simulations (100 répétitions dans l'article) et montrer des statistiques sur l'ensemble des résultats.

On commence pour les libraries qu'on va utiliser.

```
library(cluster)
library(mclust)
```

```
## Package 'mclust' version 5.4.7
## Type 'citation("mclust")' for citing this R package in publications.
```

Comme on va utiliser le Rand Index ajouté pour mesurer notre erreur, on peut définir ce fonction, qu'on appellera `rand.index()`.

```
rand.index<-function(U,V){
  n<-length(U)
```

```
  a<-0
  b<-0
  c<-0
  d<-0
  for (i in 1:(n-1)){
    for (j in (i+1):n){
      if (U[i]==U[j]){
        if (V[i]==V[j]){
          a<-a+1
          }
        else{
          b<-b+1
          }
        }
      else{
        if (V[i]==V[j]){
          c<-c+1
          }
        else{
          d<-d+1
          }
        }
      }
    }
  val<-2*(a*d-b*c)/((a+b)*(b+d)+(a+c)*(c+d))
  return(val)
}
```

On fera $N = 100$ simulations pour différents valeurs du pourcentage d'objects bruyants.

```
N<-100
k<-3
n<-120
L<-seq(0,0.30,by=0.05)
dat.cl<-c(rep("A",120),rep("B",120),rep("C",120))
#On utilise la même distribution de données en chaque groupe
total.err<-matrix(0,nrow=length(L),ncol=4)
for (l in 1:length(L)){
  t.err<-matrix(0,nrow=N,ncol=4)
  for (i in 1:N){
    #Simulation des données
    A.x<-rnorm(n,mean=0,sd=1.5)
    A.y<-rnorm(n,mean=0,sd=1.5)
    B.x<-rnorm(n,mean=6,sd=0.5)
    B.y<-rnorm(n,mean=-1,sd=0.5)
    C.x<-rnorm(n=n-as.integer(n*L[l]),mean=6,sd=0.5)
    C.y<-rnorm(n=n-as.integer(n*L[l]),mean=2,sd=0.5)
    C.xn<-rnorm(n=as.integer(n*L[l]),mean=6,sd=2)
    C.yn<-rnorm(n=as.integer(n*L[l]),mean=2,sd=2)
    C.x<-c(C.x,C.xn)
    C.y<-c(C.y,C.yn)
    dat.x<-c(A.x,B.x,C.x)
    dat.y<-c(A.y,B.y,C.y)
    dat.sim<-data.frame("x"=dat.x,"y"=dat.y,"class"=dat.cl)
```

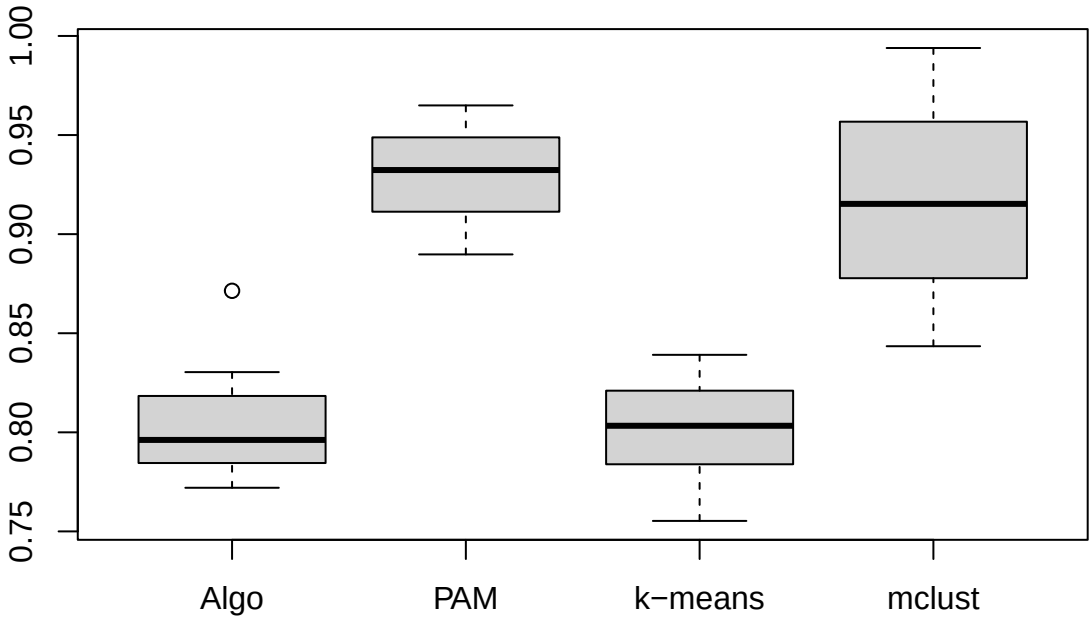
Table 1: Rand index pour différents types de clustering

% objects bruyants	Algo	PAM	k-means	mclust
0	0.8714	0.9650	0.8391	0.9939
5	0.7829	0.9520	0.7995	0.9695
10	0.8304	0.9456	0.7682	0.9439
15	0.7861	0.9324	0.8273	0.9153
20	0.7962	0.9193	0.8033	0.8929
25	0.7720	0.9033	0.8147	0.8627
30	0.8063	0.8898	0.7553	0.8434

```
    #Le algorithme dévelopé
    res.sim<-algo(dat.sim[,-3],k)
    meds<-res.sim$medoids
    clus<-res.sim$clusters
    t.err[i,1]<-rand.index(dat.cl,clus)
    #PAM
    res.sim<-pam(dat.sim[,-3],k,metric="euclidean")
    t.err[i,2]<-rand.index(dat.cl,res.sim$clustering)
    #k-means
    res.sim<-kmeans(x=dat.sim[,-3],k,nstart=1) #on peut choisir autre nstart
    t.err[i,3]<-rand.index(dat.cl,res.sim$cluster)
    #mclust
    res.sim<-Mclust(dat.sim[,-3],G=3,verbose=FALSE)
    t.err[i,4]<-rand.index(dat.cl,res.sim$classification)
  }
  total.err[l,]<-colMeans(t.err)
}
total.err<-cbind(L*100,total.err)
colnames(total.err)<-c("% objects bruyants","Algo","PAM","k-means","mclust")
knitr::kable(total.err,format="latex",
             caption = "Rand index pour différents types de clustering",digits = 4)
```

```
boxplot(total.err[,-1],main="Boîte à moustaches dus rand.index pour des données simmulées")
```

Boîte à moustaches dus rand.index pour des données simmulées



On peut voir que dans l'ensemble des cas considérés l'algorithme dévelopé a un rand index assez pire que les autres algorithmes sauf le k-means, où on considère uniquement une initialisation. Mais, pour le k-means, si on fixe plus d'une initialisation on peut obtenir un résultat meilleur. Il faut remarquer l'existance d'un résultat remarquablement bon d'un rand index pour notre algorithme.

```
L[which.max(total.err[,2])]
```

```
## [1] 0
```

Ce résultat se correspond au cas où il n'y a pas des données bruyants.

Données `iris`

1. Appliquer votre algorithme au jeu de données iris et comparer aux k-means et à l'algorithme `PAM` du package `cluster` de `R`, et `mclust`.

On va appliquer notre algorithme au jeu de données `iris` qui contient données de trois espèces de fleurs. Alors, on va l'initialiser avec $k = 3$.

Après, on fera matrice de confussion:

```
dat<-iris
noms<-levels(dat$Species)
summary(dat)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
k<-3
calc<-algo(dat[,-5],k)
```

```
medoids<-calc$medoids
groupes<-calc$clusters
M<-matrix(0,nrow=k,ncol=k)
noms<-dat[medoids,]$Species
for (i in 1:k){
  for (j in 1:k){
    M[i,j]<-sum(dat[groupes==i,]$Species==noms[j])
  }
}
colnames(M)<-noms
rownames(M)<-noms
knitr::kable(total.err,format="latex",caption =
               "Comparaison des rand index obtenues pour $N=100 simulations
               en utilisant différents algorithmes",digits = 4)
```

\begin{table}

\caption{Comparaison des rand index obtenues pour $N=100 simulations en utilisant différents algorithmes}

% objects bruyants	Algo	PAM	k-means	mclust
0	0.8714	0.9650	0.8391	0.9939
5	0.7829	0.9520	0.7995	0.9695
10	0.8304	0.9456	0.7682	0.9439
15	0.7861	0.9324	0.8273	0.9153
20	0.7962	0.9193	0.8033	0.8929
25	0.7720	0.9033	0.8147	0.8627
30	0.8063	0.8898	0.7553	0.8434

\end{table}

```
rand.index(dat$Species,groupes)
```

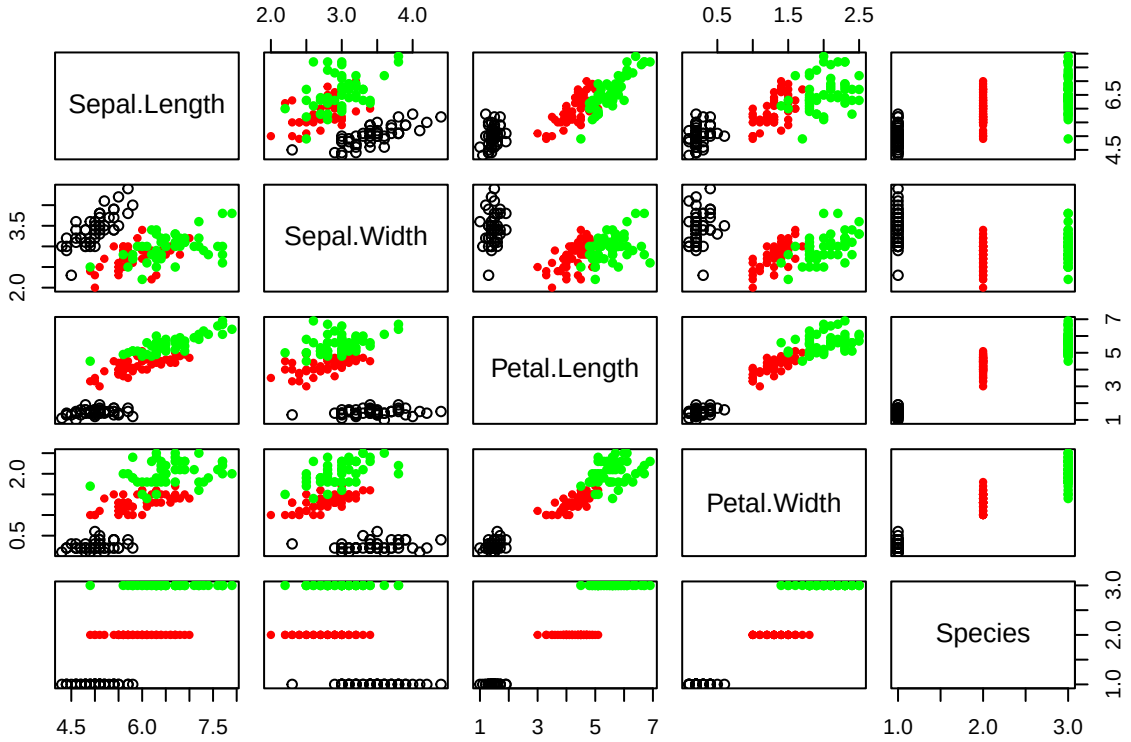
```
## [1] 0.8342589
```

On peut voir dans le tableaux qu'on a bien classifié 136 sur 150 données. De plus, le rand index obtenu est 0.8342589.

2. Visualiser les différentes partitions sur les premiers plans d'une analyse en composantes principales.

En premier lieu, on va faire une analyse de la corrélation entre variables.

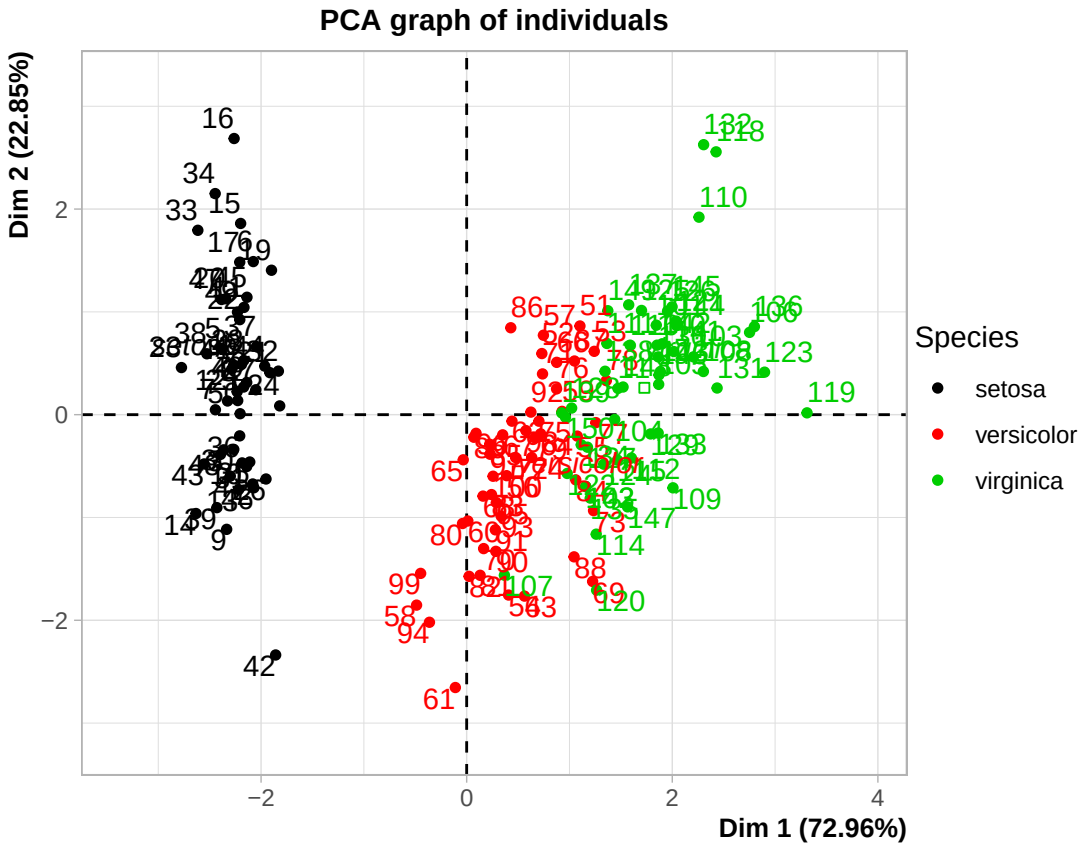
```
pairs(iris,col=c("black","red","green")[iris$Species],pch=c(21,20,16)[iris$Species])
```



On peut voir dans la figure que certaines variables comme la longueur ou la largeur du pétale sont bien différencies pour l'espéce setosa en relation aux autres. Donc, on peut sentir que on aura aussi ce séparation dans la répresentation graphique de notre PCA.

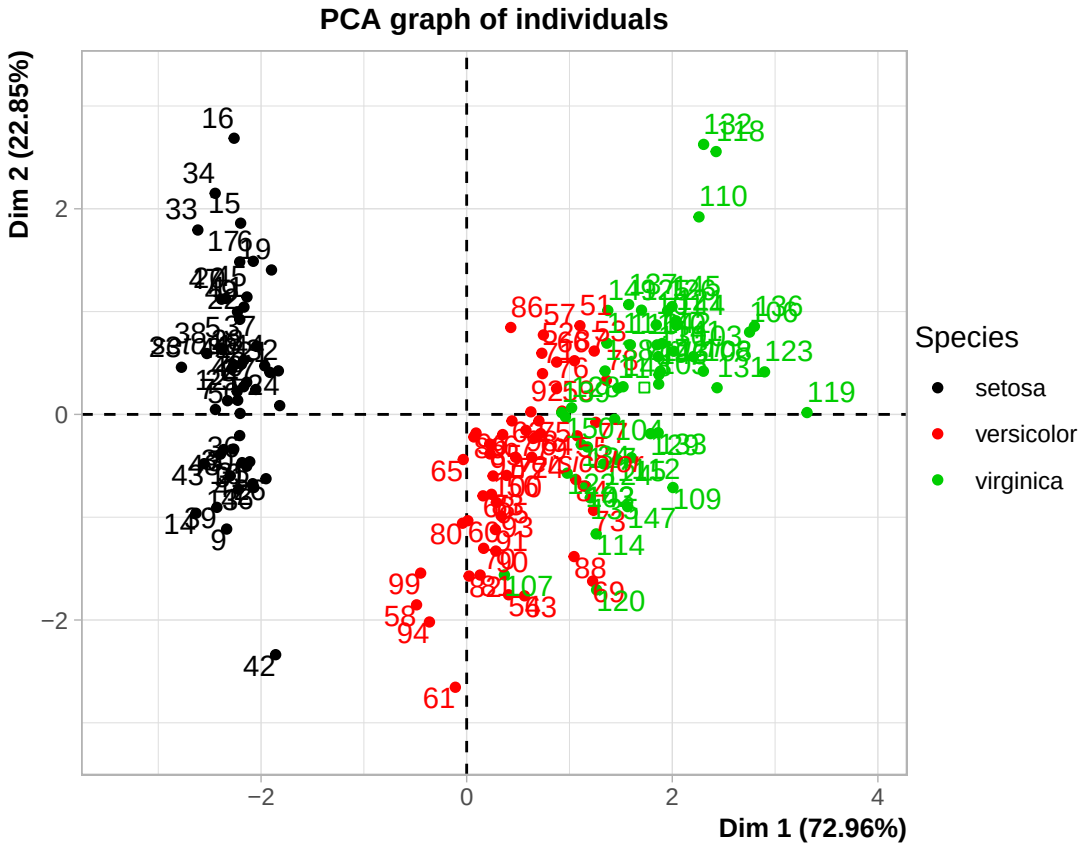
On utilisera la fonction `PCA()` de `FactoMineR` pour faire l'analyse en composantes principales. Dans ce cas, on va escalader des données car l'échelle de la longueur est plus grande que l'échelle de la largueur.

```
library(FactoMineR)
res.pca<-PCA(iris,scale.unit=TRUE,ncp=5,graph=FALSE,quali.sup=5)
plot(res.pca,habillage=ncol(iris),choix="ind")
```



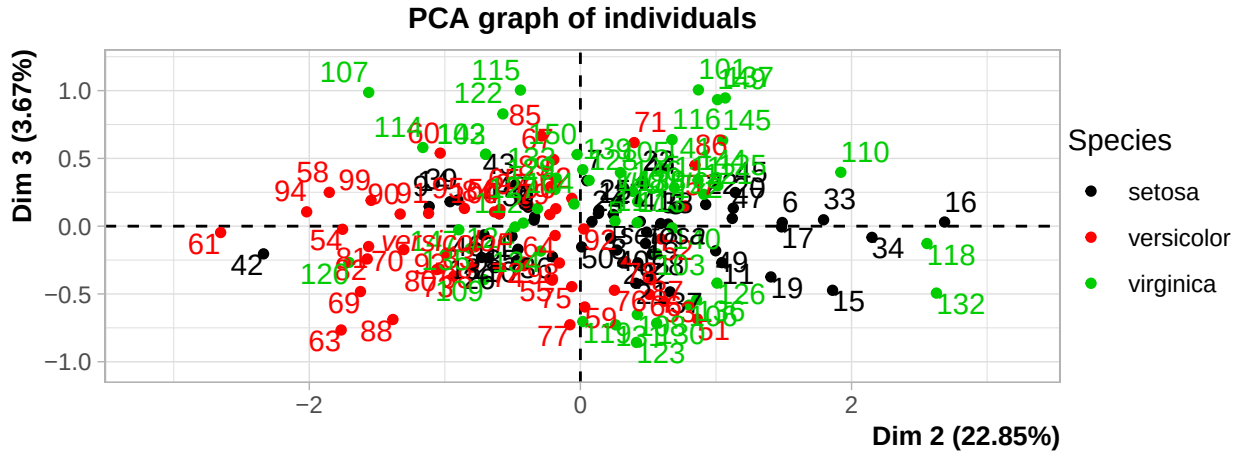
Comme on peut voir dans la figure, la première composante peut différencier les setosa des autres espèces. De plus, on peut voir dans la figure que les données de l'espèce versicolor sont un peu plus à gauche dans la premier composante. Mais, on ne peut pas être capables de faire une distintion entre les deux espèces. Pour la deuxième composante, on n'a pas une distinction entre les espèces. Alors, si on considère le plan complète, la classification qu'on peut faire est sa qui se corresponde à la première composante principale.

```
iris2<-iris
for (i in 1:k){
  iris2[groupes==i,]$Species<-noms[i]
}
plot(res.pca,habillage=ncol(iris2),choix="ind")
```

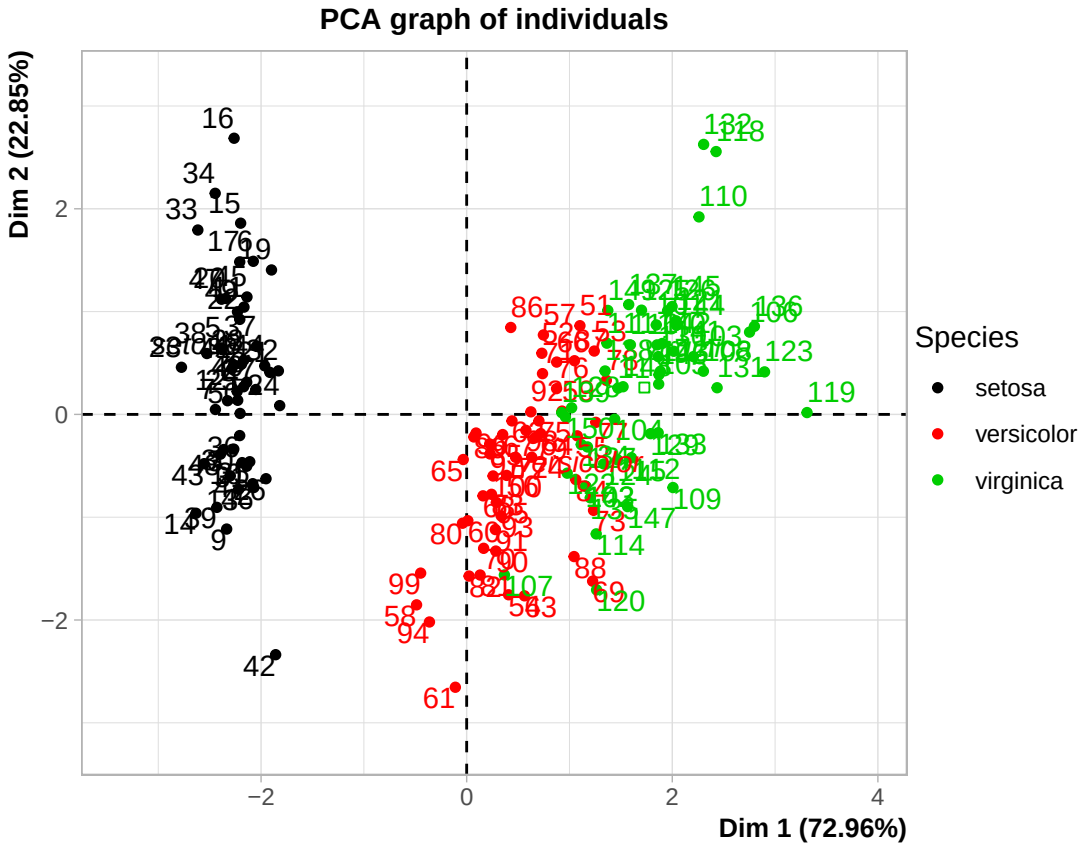


Comme pour le PCA on ne prend pas que les variables quantitatives, et on fait le clustering pour une variable qualitative, on sait qu'on obtient le même analyse en composantes principales pour la partition obtenue avec notre algorithme. Dans ce cas, on peut faire une conclusion similaire à ce qu'on a fait pour la partition réelle. La première composante nous permet de différencier le groupe setosa des autres. Mais, la deuxième composante ne nous permet pas de faire une classification entre nos groupes. De plus, si on regarde le plan général, on a que la classification se corresponde à la qu'on peut obtenir de la première composante.

```
plot(res.pca,axes=c(2:3),habillage=ncol(iris),choix="ind")
```




```
plot(res.pca,habillage=ncol(iris2),choix="ind")
```



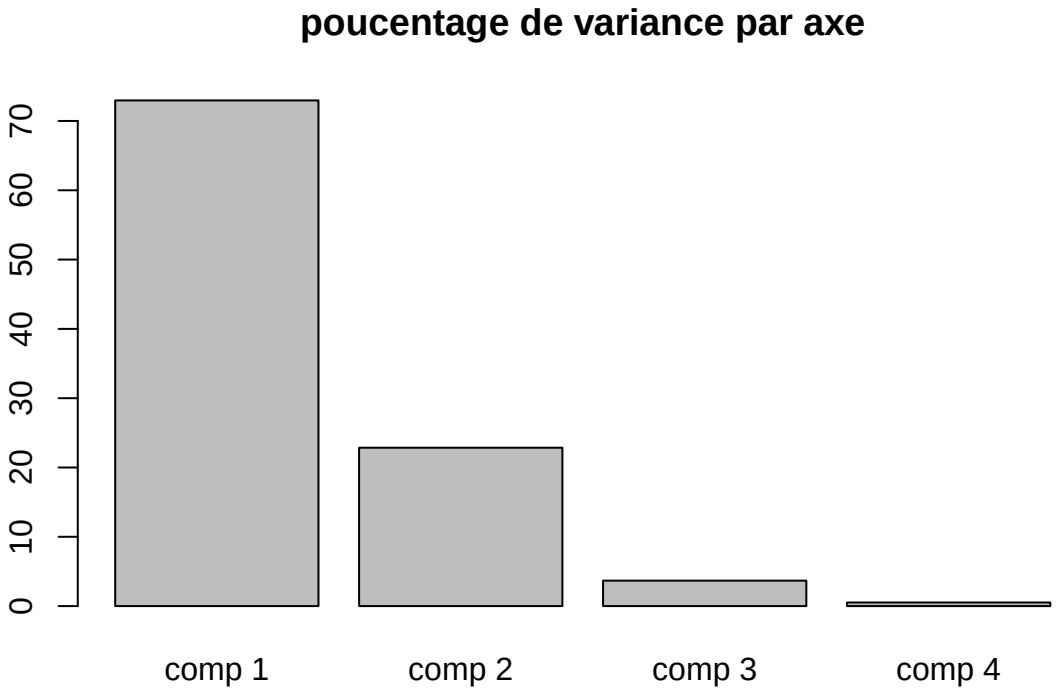
Dans le cas du plan qui représente la deuxième et la troisième composantes principales, on ne peut pas faire une séparation entre nos partitions, soit la réelle soit l'obtenue à partir de notre algorithme.

On va faire une analyse plus détaillée de notre PCA.

```
eigvalues<-data.frame(res.pca$eig)
res.pca$eig
```

```
##        eigenvalue percentage of variance cumulative percentage of variance
## comp 1 2.91849782             72.9624454                           72.96245
## comp 2 0.91403047             22.8507618                           95.81321
## comp 3 0.14675688              3.6689219                           99.48213
## comp 4 0.02071484              0.5178709                          100.00000
```

```
barplot(eigvalues$percentage.of.variance,names.arg=row.names(eigvalues),
main='poucentage de variance par axe')
```

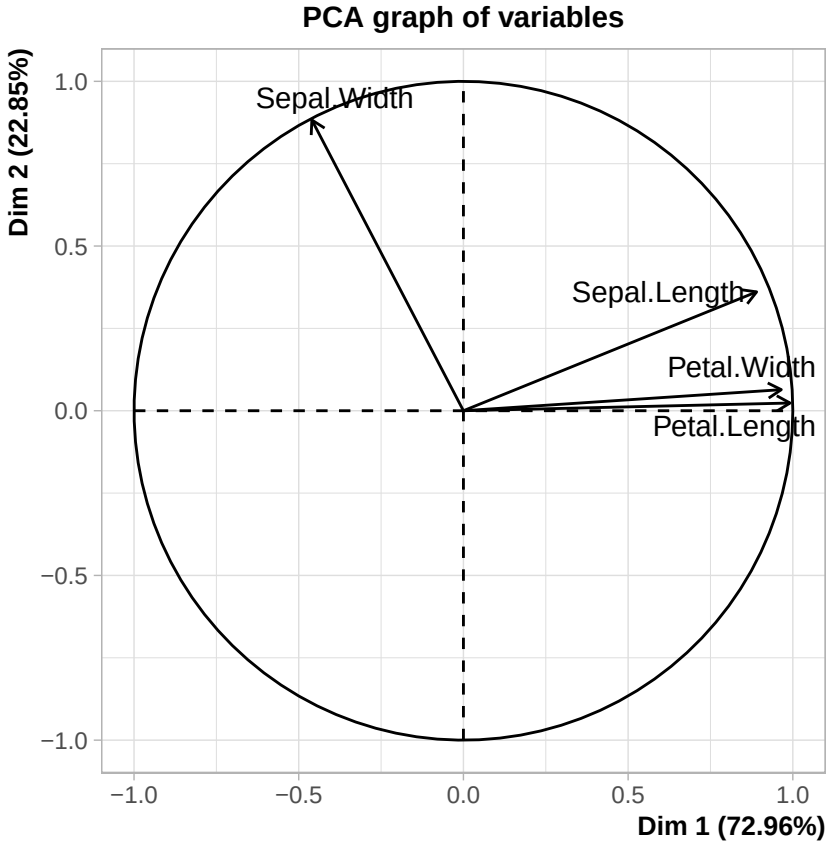


On peut voir que les deux premiers composantes expliquent 95.81321% de l'inertie et les tres premiers composantes expliquent un 99.48213% de l'inertie Donc, le nuage initial est pratiquement dans un espace de dimension 3.

```
res.pca$var$contrib
```

```
##                  Dim.1       Dim.2     Dim.3     Dim.4
## Sepal.Length 27.150969 14.24440565 51.777574  6.827052
## Sepal.Width   7.254804 85.24748749  5.972245  1.525463
## Petal.Length 33.687936  0.05998389  2.019990 64.232089
## Petal.Width  31.906291  0.44812296 40.230191 27.415396
```

```
plot(res.pca,choix="varcor",axes=1:2)
```



On peut voir que pour la premier composante, elle est composée principalement pour la longueur du sépale et la longuer et la largueur du pétale en parts égales.De plus, elle est positivamente corrélée à ces variables. Après, la deuxiéme composante principale est composée pour la largueur du sépale principalement et un peu pour la longueur du sépale. Elle est positivament corrélée aux deux variables.

3. Commentez.

Il se vérifie que l'algorithme codé, pour le même ensemble des données, doit faire toujours les mêmes clusters. Malgrè ce fait, on peut remarquer qu'on obtient un meilleur résultat que l'obtenu par les autheurs du article pour le cas des données `iris` mais pire pour les données simulées. De plus, en comparaison avec des autres algorithmes, les rand index obtenus sont assez pires sauf pour le `k-means`. Mais il faut remarquer que la bonté des résultats du `k-means` dépend du nombre de initialisations considérées. Donc, si on fait un nombre plus grand d'initialisations ça peut impliquer une meilleure classification.

On peut considérer soit que le nombre de cas considéré n'est pas assez large soit qu'il y a un erreur dans le code qu'implique des valeurs différents. Mais, en tout cas on a développé un algortihme qui classifié correctement les données iris.