TP no 1 Régression linéaire & polynomiale, ACP et gaussiennes multivariées - solution

UE Apprentissage Statistique

1 Exercice 1

Dans cet exercice, on dispose d'une variable d'entrée, X, d'une variable de sortie Y et on veut effectuer la régression de Y en fonction de X à partir d'un échantillon de n points $(x_i, y_i), i = 1, ..., n$.

Le modèle de la régression linéaire est le suivant :

$$Y = \beta_0 + \beta_1 X + \epsilon,$$

où ϵ est une erreur résiduelle, qu'on considère souvent suivre la loi $\mathcal{N}(0, \sigma^2)$.

La fonction que l'on cherche à estimer est donc $f(X) = \beta_0 + \beta_1 X$ et on l'estime par les moindres carrés à partir de notre échantillon :

$$(\hat{\beta}_0, \hat{\beta}_1) = \text{argmin}_{\beta_0, \beta_1} \sum_{i=1}^{n} (y_i - f(x_i))^2.$$

En `R` cette procédure est réalisée par la fonction `lm`:

```
load("regression-dataset.Rdata")
fit  = lm(y ~ x)
```

L'objet renvoyé par la fonction `lm` contient de nombreuses variables et notamment :

- `fit$coefficients` : les coefficients estimés $\hat{\beta}_0$ et $\hat{\beta}_1$.
- `fit$fitted.values` : les valeurs obtenues par le modèle sur les données d'entrainement, i.e.,

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$$

- `fit$residuals` : les résidus obtenus sur l'échantillon d'entrainement, i.e., les valeurs $(y_i - \hat{y}_i)$.

Par ailleurs, on peut obtenir un "résumé" du modèle obtenu en utilisant la fonction `summary` :

```
summary(fit)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -30.9726  -9.5429   0.3133   9.2285  29.2748
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -8.6672     3.2665  -2.653   0.0103 *
```

```
## x             9.8235     0.6087  16.140   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 14.36 on 58 degrees of freedom
## Multiple R-squared:  0.8179, Adjusted R-squared:  0.8147
## F-statistic: 260.5 on 1 and 58 DF,  p-value: < 2.2e-16
```

Ce "résumé" contient notamment les coefficients, et leurs p-valeurs associées.

La fonction `predict` permet d'obtenir les prédictions du modèle sur de nouvelles valeurs de x. Il suffit pour cela utiliser l'option `newdata`. Il faut néanmoins prendre soin de préciser qu'on renseigne bien la variable `x` (et plus généralement le nom de la variable que l'on a utilisée pour construire le modèle):

```
#preds = predict(fit, newdata = x.test) # does not work
preds = predict(fit, newdata = data.frame("x"=x.test))
```

Enfin, on peut visualiser le modèle obtenu en passant directement l'objet renvoyé par la fonction `lm` à la fonction `abline` (qui permet de tracer des droites de type $y = ax + b$).

Le modèle de la régression polynomiale consiste à estimer une fonction non linéaire entre x et y en considérant des transformations polynomiales de x. Dans le cas quadratique, on estime la fonction :

$$f(X) = \beta_0 + \beta_1 X + \beta_2 X^2$$

et plus généralement, à l'ordre d :

$$f(X) = \beta_0 + \sum_{i=1}^{d} \beta_i X^i$$

On est dans le cas d'une régression linéaire multivariée où les différentes variables résultent de la transformation de la variable d'origine. On peut donc estimer les coefficients β_i par moindres carrés en utilisant la fonction `lm`.

En pratique, c'est en réalité très simple car plutôt que d'avoir à explicitement construire les variables $X^2, X^3, ..., X^d$, on peut directement utiliser la fonction `poly` dans l'appel à la fonction `lm` :

```
fit3 = lm(y ~ poly(x,3,raw=TRUE))
fit3$coefficients
```

```
##             (Intercept) poly(x, 3, raw = TRUE)1 poly(x, 3, raw = TRUE)2
##              15.6344521             -12.4929297               3.9989189
## poly(x, 3, raw = TRUE)3
##              -0.1891172
```

```
# more details with summary(fit3)
```

Notons néanmoins qu'il convient de préciser l'option `raw=TRUE` pour obtenir l'expansion polynomiale "classique" décrite ci-dessus. Par défaut, la fonction `poly` réalise en effet une orthogonalisation de ces polynomes, ce qui peut être en pratique plus robuste et constitue donc une bonne stratégie. C'est sans grande importance pour cet exercice et nous considèrerons la transformation classique.

De la même manière, pour réaliser la prédiction, il suffit de passer les nouvelles observations x, les expansions polynomiales sont réalisées automatiquement :

```
preds3 = predict(fit3, newdata = data.frame("x"=x.test))
```

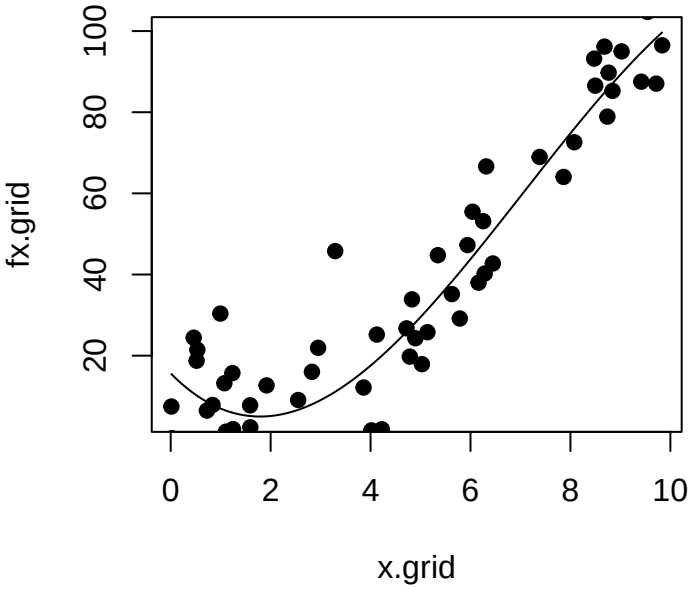
En revanche, le modèle ne se traduisant plus par une droite $y = ax + b$, on ne peut plus le tracer en utilisant la fonction `abline`. Pour la visualiser, deux possibilités :

- extraire les coefficients du modèle et utiliser la fonction `curve`

- discrétiser la gamme de valeurs de x considérée (e.g., celle de l'échantillon) et obtenir les valeurs prédites par le modèle par la fonction `predict`

Cette seconde solution peut s'implémenter ainsi :

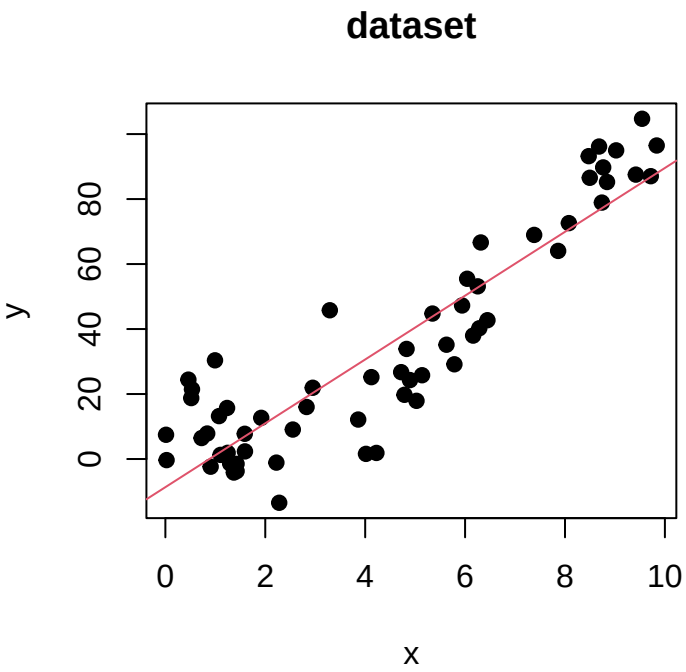
```
x.grid = seq(min(x),max(x),by=0.01)
fx.grid = predict(fit3, newdata = data.frame("x"=x.grid))
plot(x.grid, fx.grid, type = "l")
points(x, y, pch = 19)
```



1.1 Question 1 et 2

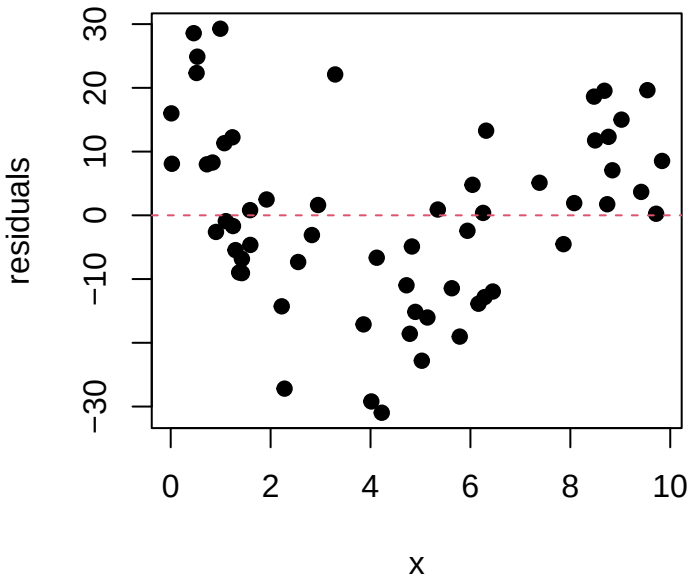
On effectue la régression linéaire :

```
##############################
#### STARTING EXERCICE 1 ####
##############################
# load dataset
load("regression-dataset.Rdata")
# plot
plot(x, y, pch = 19, main = "dataset")
fit  = lm(y ~ x)
abline(fit, col = 2)
```



On observe que le modèle n'est pas très satisfaisant car les résidus $y_i - \hat{y}_i$ semblent plutôt positifs pour les valeurs faibles et élevées de x, et plutôt négatifs pour les valeurs intermédiaires. Ceci apparait clairement quand on trace la valeur des résidus en fonction de x :

```
plot(x, fit$residuals, xlab = "x", ylab = "residuals", pch = 19)
abline(h=0, lty = 2, col = 2)
```



Si le modèle linéaire était adapté les résidus seraient distribués de manière homogène autour de 0.

1.2 Question 3

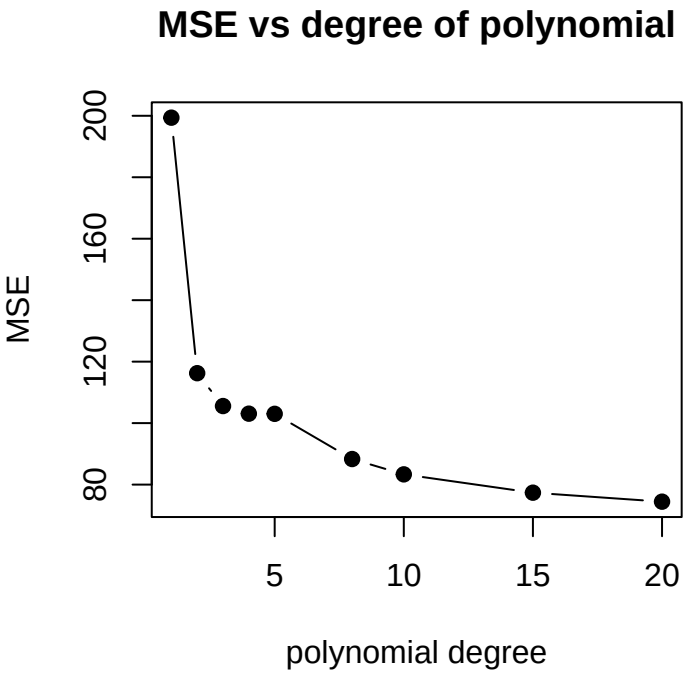
Pour calculer l'erreur obtenue sur les données utilisées pour apprendre le modèle, on peut directement utiliser le champ `fitted.values` de l'objet `lm` (voire même directement le champ `residuals`).

```
# specify polynomial degrees to consider
deg.list = c(1:5,8,10,15,20)
```

```r
# compute MSE
MSE = c()
for(d in deg.list){
    fit = lm(y ~  poly(x,d,raw=TRUE))
    mse = mean( (y - fit$fitted.values)^2 )
    cat("*** degree", d, ": MSE on training data =", mse, "***\n")
    MSE = c(MSE, mse)
}
```

```
## *** degree 1 : MSE on training data = 199.3832 ***
## *** degree 2 : MSE on training data = 116.2499 ***
## *** degree 3 : MSE on training data = 105.556 ***
## *** degree 4 : MSE on training data = 103.0742 ***
## *** degree 5 : MSE on training data = 103.0232 ***
## *** degree 8 : MSE on training data = 88.31346 ***
## *** degree 10 : MSE on training data = 83.33188 ***
## *** degree 15 : MSE on training data = 77.36289 ***
## *** degree 20 : MSE on training data = 74.44325 ***
```

```r
# plot
plot(deg.list, MSE, type ="b", xlab = "polynomial degree", ylab = "MSE", main = "MSE vs degree of polyno
```



On observe que l'erreur d'apprentissage décroît à mesure que le degré du polynome augmente.

1.3 Question 4

Pour calculer l'erreur sur de nouvelles données, il faut passer par la fonction `predict` en utilisant l'argument `newdata`.

```r
MSE.test = c()
for(d in deg.list){
  fit = lm(y ~ poly(x,d,raw=TRUE))
  preds = predict(fit, newdata = data.frame("x"=x.test))
  mse = mean( (y.test - preds)^2 )
```

```
  cat("*** degree", d, ": MSE on test data =", mse, "***\n")
  MSE.test = c(MSE.test, mse)
}
```

```
## *** degree 1 : MSE on test data = 193.7339 ***
## *** degree 2 : MSE on test data = 147.3679 ***
## *** degree 3 : MSE on test data = 142.8611 ***
## *** degree 4 : MSE on test data = 146.4325 ***
## *** degree 5 : MSE on test data = 146.5772 ***
## *** degree 8 : MSE on test data = 179.3409 ***
## *** degree 10 : MSE on test data = 170.5381 ***
```

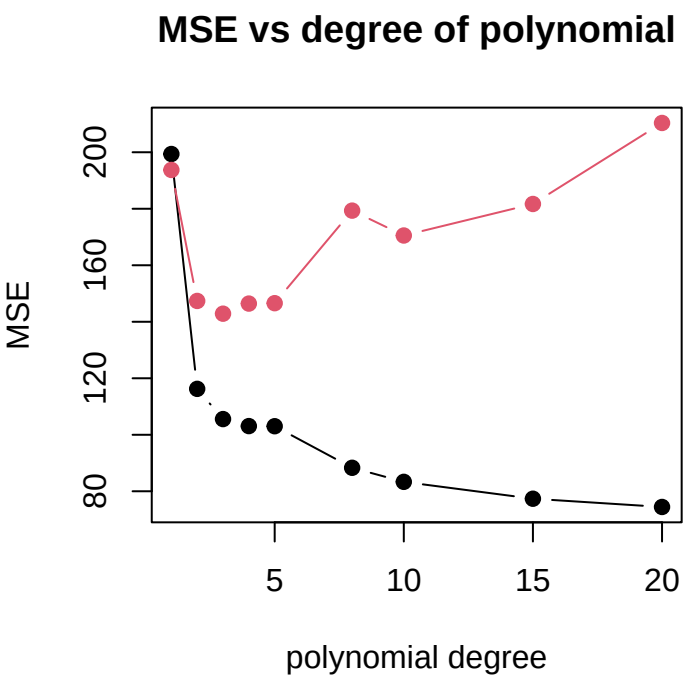
```
## Warning in predict.lm(fit, newdata = data.frame(x = x.test)): prediction from
## rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

```
## *** degree 15 : MSE on test data = 181.6938 ***
```

```
## Warning in predict.lm(fit, newdata = data.frame(x = x.test)): prediction from
## rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

```
## *** degree 20 : MSE on test data = 210.3687 ***
```

```
# plot
plot(deg.list, ylim = range(c(MSE,MSE.test)), MSE, type ="b", xlab = "polynomial degree", ylab = "MSE",
lines(deg.list, MSE.test, type = "b", col = 2, pch = 19)
```



On constate que contrairement à ce qu'on observe sur les données d'apprentissage, l'erreur sur les nouvelles données commence par décroître jusqu'à un degré égal à 3, ce qui confirme que le modèle linéaire n'était pas optimal, mais augmente pour des degrés plus élevés. C'est un phénomène de sur-apprentissage.

1.4 Question 5

La figure suivante permet de visualiser l'ensemble des modèles obtenus (en haut à gauche), ainsi que les modèles de degré 3, 10 et 20. L'effet du sur-apprentissage est clair : pour des degrés élevés du polynome, le modèle capture des spécificités des données d'apprentissage.

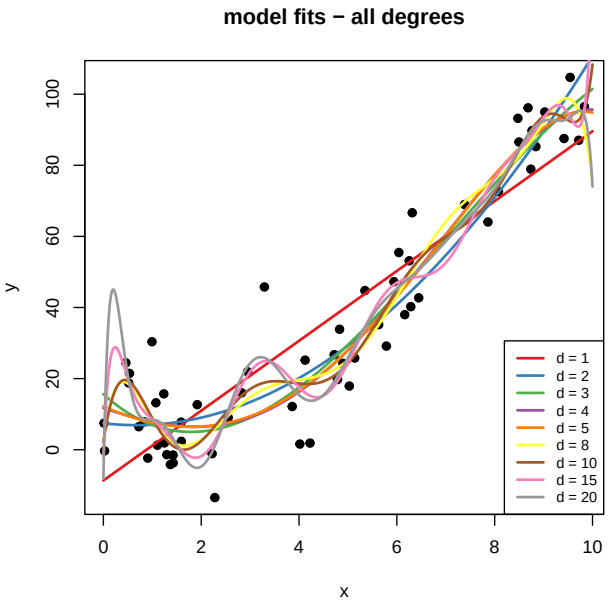
```r
# define  colors
library(RColorBrewer)
cols = brewer.pal(length(deg.list), "Set1")
# define grid of x values
x.grid = seq(0,10,by=0.01)
# define mfrow
par(mfrow = c(2,2))
# plot models  - all on the same graph
plot(x,y, pch = 19, main = "model fits - all degrees")
for(i in seq(length(deg.list))){
  d = deg.list[i]
  fit = lm(y ~ poly(x,d,raw=TRUE))
  preds = predict(fit, newdata = data.frame("x"=x.grid))
  lines(x.grid, preds, col = cols[i], lwd = 2)
}
```

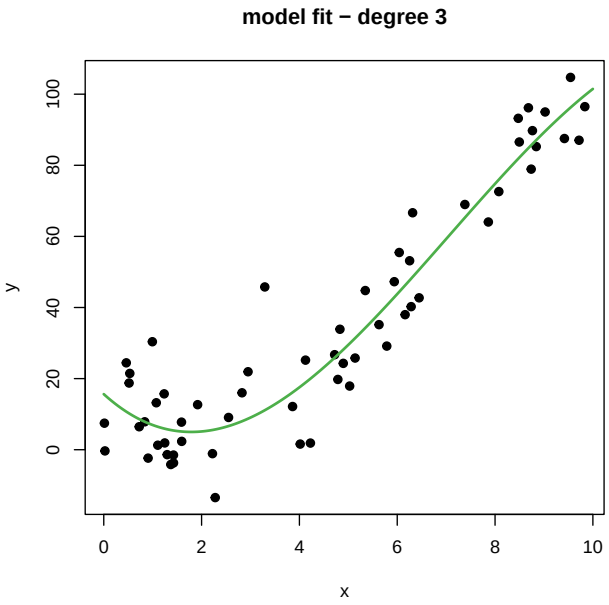
```
## Warning in predict.lm(fit, newdata = data.frame(x = x.grid)): prediction from
## rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

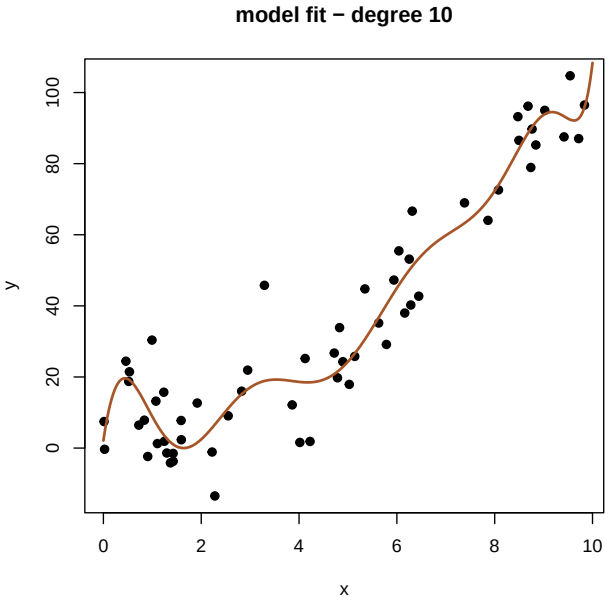
```
## Warning in predict.lm(fit, newdata = data.frame(x = x.grid)): prediction from
## rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

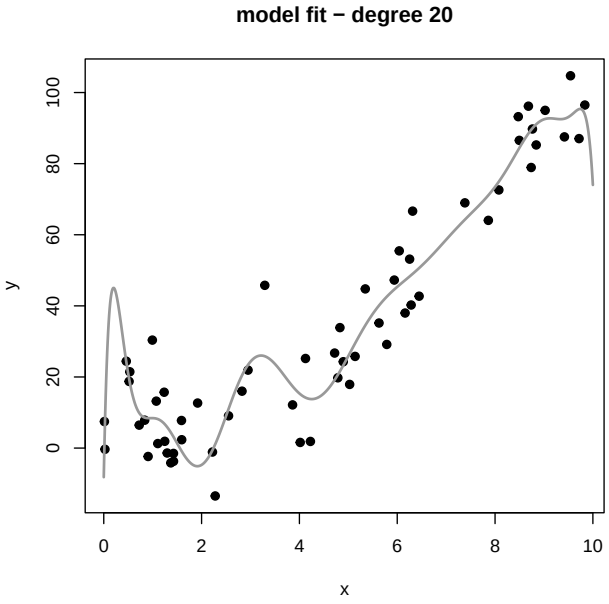
```r
legend("bottomright", paste("d =",deg.list), col = cols, lwd = 2, cex = 0.8)
# plot intermediate models
deg.list.small = c(3,10,20)
for(d in deg.list.small){
  fit = lm(y ~ poly(x,d,raw=TRUE))
  preds = predict(fit, newdata = data.frame("x"=x.grid))
  plot(x, y, pch = 19, main = paste("model fit - degree", d))
  lines(x.grid, preds, col = cols[which(deg.list==d)], lwd = 2)
}
```

```
## Warning in predict.lm(fit, newdata = data.frame(x = x.grid)): prediction from
## rank-deficient fit; attr(*, "non-estim") has doubtful cases
```





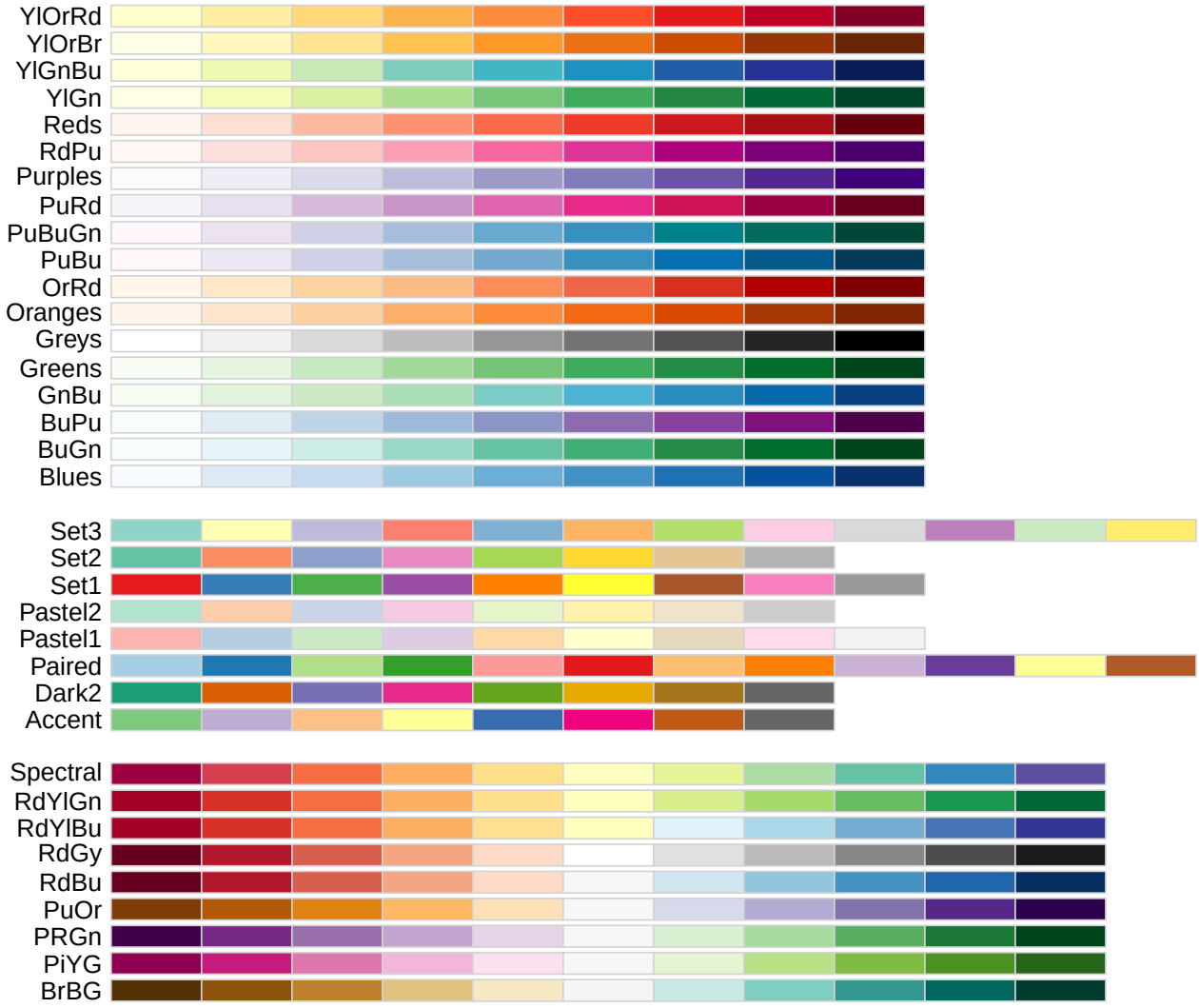





```
par(mfrow = c(1,1))
```

Enfin, une remarque liée à la gestion / au choix de couleurs en `R`. Le package `RColorBrewer` propose un certain nombre de "palettes" pré-définies qui sont bien pratiques. Pour obtenir un ensemble de n couleurs d'une palette donnée, il suffit d'appeler la fonction `brewer.pal(n,name)`, où `name` est le nom de la palette en question. La figure ci-dessous représente l'ensemble des palettes disponibles.

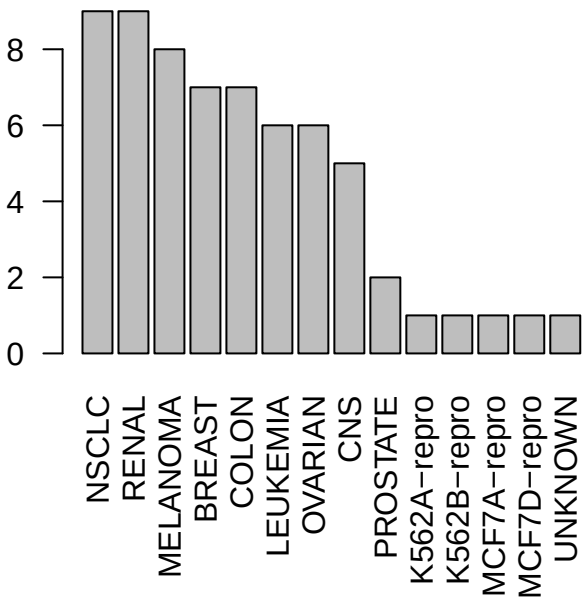
```
display.brewer.all()
```



2 Exercice 2

2.1 Question 1

```
############################
#### STARTING EXERCICE 2 ####
############################
y = as.character(read.table("datasets/nci.label")$V1)
X = read.table("datasets/nci.data")
X = t(X)
tt = sort(table(y), decreasing = TRUE)
par(mar = c(7,4,4,2))
barplot(tt, main  = "number of observations per class", las = 2)
```




```
par(mar = c(5,4,4,2))
```

2.2 Question 2

```
ind1 = grep("repro", y)
ind2  = which(y == "UNKNOWN")
ind.rm = c(ind1, ind2)
y = y[-ind.rm]
y = factor(y)
X = X[-ind.rm,]
```

2.3 Question 3

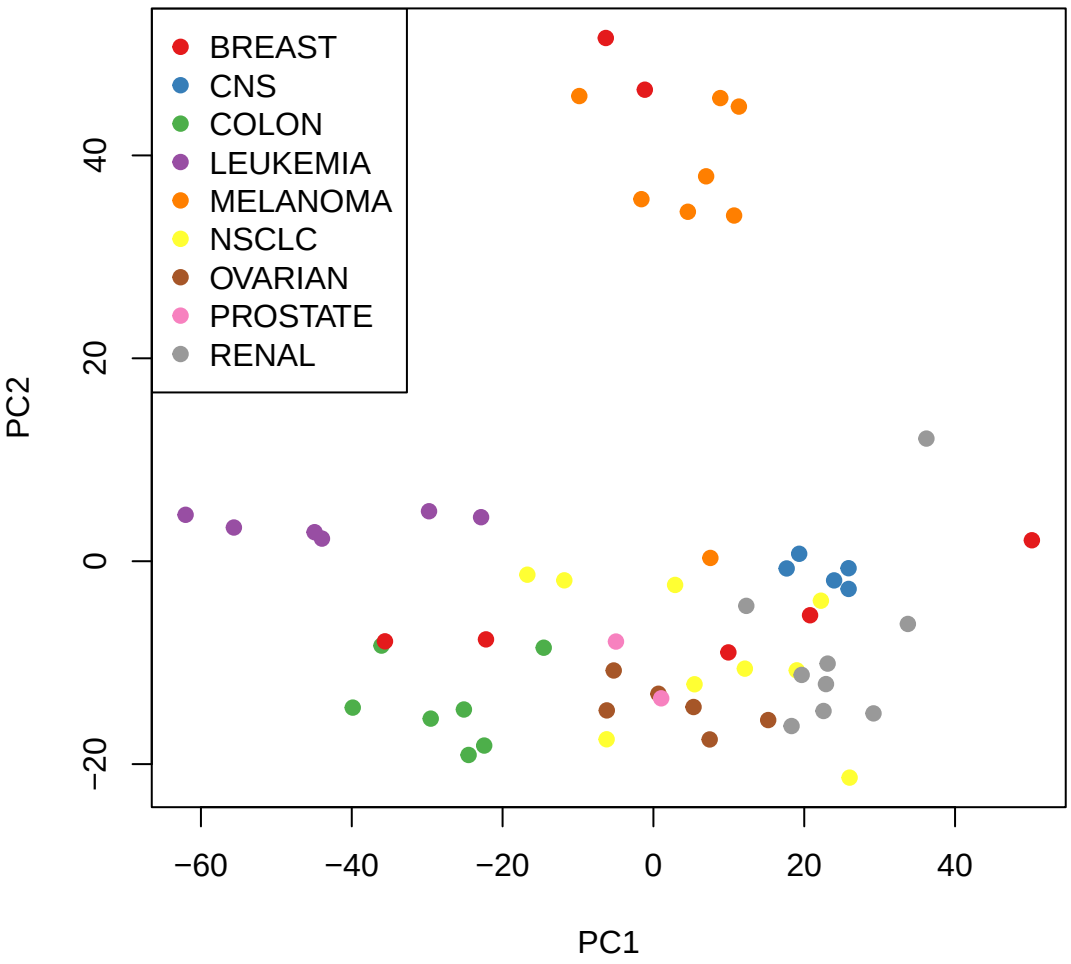
```
pca = prcomp(X)
cat("*** number of PCs =", ncol(pca$x), "***\n")
```

```
## *** number of PCs = 59 ***
```

On trouve 59 composantes principales, comme attendu : le nombre de composantes est égal à la valeur minimale du nombre d'échantillons et de variables.

2.4 Question 4

```
# define  colors
cols = brewer.pal(nlevels(y), "Set1")
# plot
plot(pca$x[,1], pca$x[,2], pch = 19, col = cols[y], xlab = "PC1", ylab = "PC2", main = "Firt two PCs")
legend("topleft", levels(y), col = cols, pch = 19)
```

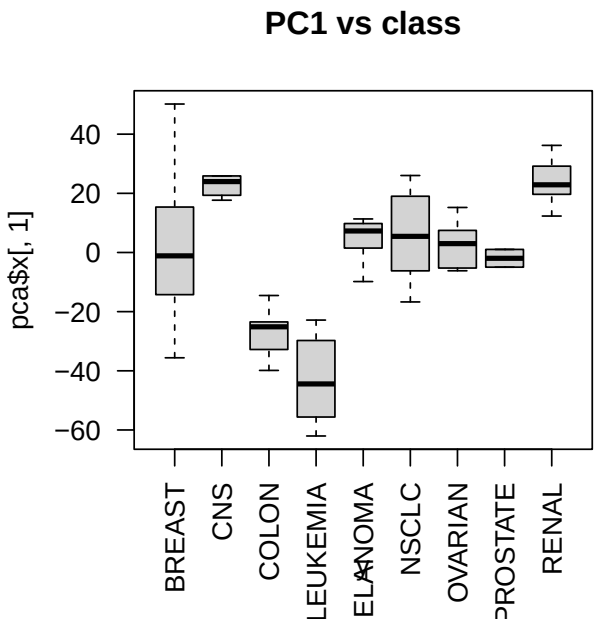


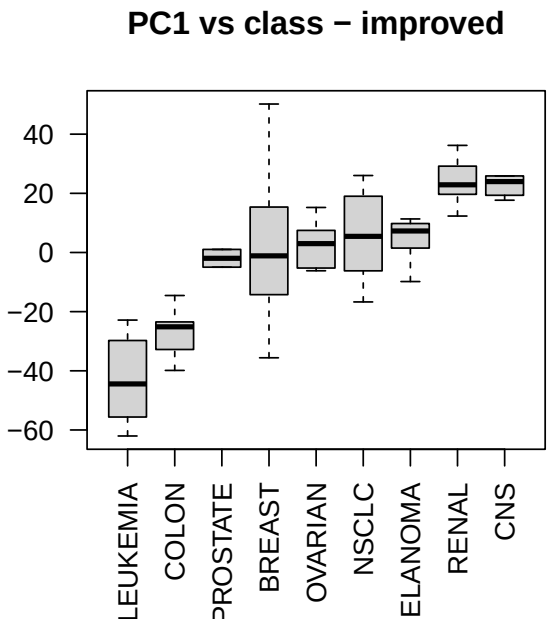
On note que les tumeurs tendent à s'organiser – bien qu'imparfaitement – en fonction de leur origine selon le premier axe. Le deuxième axe quand à lui sépare clairement la plupart des mélanomes (sauf 1) ainsi que deux cancers du sein du reste du jeu de données.

2.5 Question 5

Pour conforter ces observations, on peut tout simplement tracer la valeur des composantes principales en fonction du type de cancer. La figure suivante s'intéresse à la 1ère composante, le graphe de droite étant le même que celui de gauche après un ré-ordonancement pour rendre l'effet plus clair.

```
par(mfrow = c(1,2))
# plot
boxplot(pca$x[,1] ~ y, main = "PC1 vs class", las = 2)
# improved plot
tt = split(pca$x[,1], y)
ind.sort = order( sapply(tt, median) )
tt = tt[ind.sort]
boxplot(tt, main = "PC1 vs class - improved", las = 2)
```

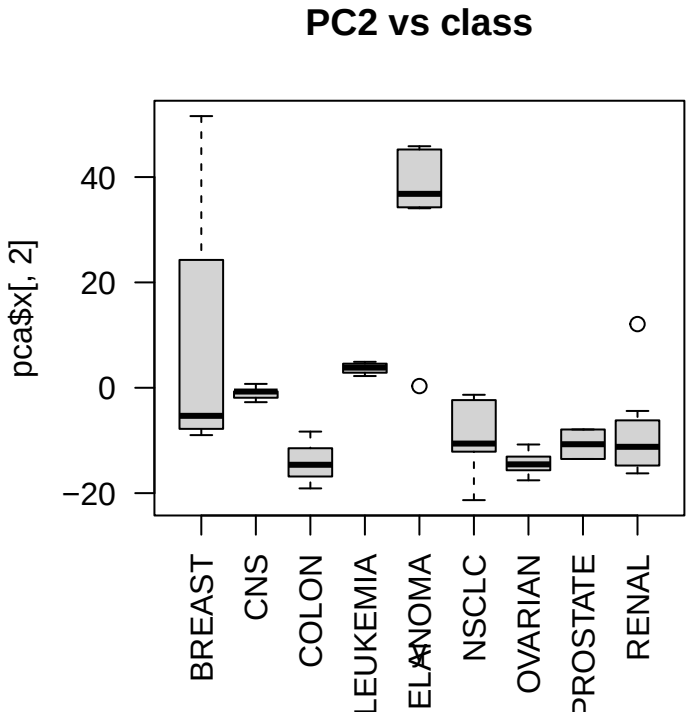





```
par(mfrow = c(1,1))
```

La figure suivante s'intéresse à la 2nde composante.

```
boxplot(pca$x[,2] ~ y, main = "PC2 vs class", las = 2)
```



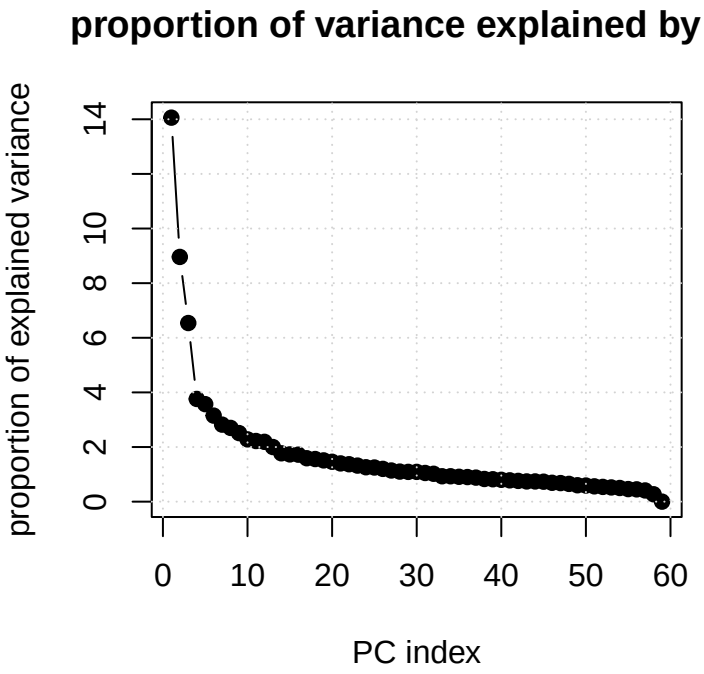
2.6 Question 6

Le champ `sdev` de l'objet rendu par la fonction `prcomp` renvoie l'écart type de chacune des composantes principales[1]. Le code ci-dessous convertit cette grandeur en proportion de variance expliquée et fournit le

[1]NB : on peut également calculer cet écart-type selon les colonnes de la matrice x de l'objet rendu par `prcomp`.

graphe de la proportion de variance expliquée en fonction de l'indice de la composante principale. On appelle parfois ce graphe un "scree plot" (scree comme éboulis).

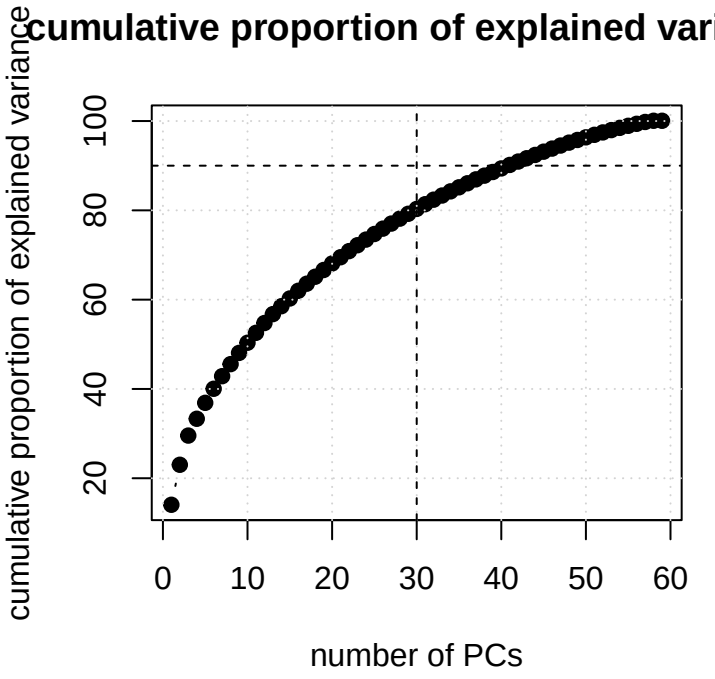
```
# compute and cast to percent
prop.var = pca$sdev^2
prop.var = prop.var  / sum(prop.var)
prop.var = round(100*prop.var, digits = 2)
# plot "scree"
plot(prop.var, type = "b", pch = 19, xlab = "PC index", ylab = "proportion of explained variance", main
grid()
```



2.7 Question 7

Enfin, la figure suivante représente la proportion de variance expliquée cumulée, i.e., quand on conserve les premières composantes principales. On note que pour conserver de l'ordre de 90% de variance, on peut se restreindre au 40 premières composantes principales. Par rapport aux 6380 variables initiales, cela représente une réduction d'un facteur 170. A l'inverse, si on souhaite conserver 80% de la variance totale, on peut se restreindre au 30 premières composantes.

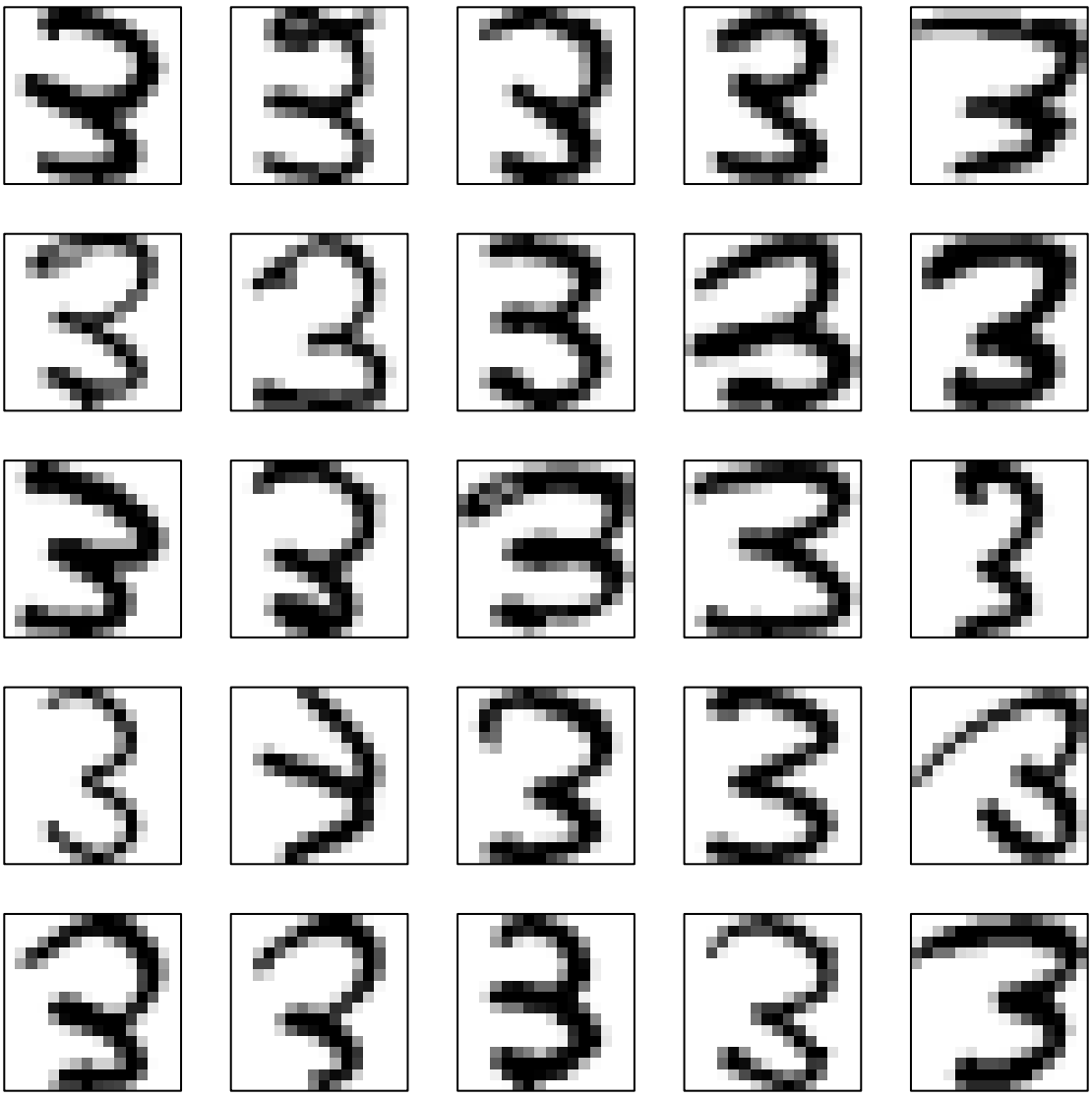
```
# plot cumulative
plot(cumsum(prop.var), type = "b", pch = 19, xlab = "number of PCs", ylab = "cumulative proportion of ex
grid()
abline(h = 90, lty = 2)
abline(v = 30, lty = 2)
```



3 Exercice 3

3.1 Question 1

```
##############################
#### STARTING EXERCICE 3 ####
##############################
rm(list = ls())
load("datasets/digits-3.Rdata")
# define colors
cols = gray(seq(1,0,length.out=256))
# show a few images
n = 5
par(mfrow = c(n,n))
par(mar = c(1,1,1,1))
set.seed(27)
ind.sple = sample(dim(I)[3], n*n)
for(i in 1:length(ind.sple)){
  image(I[,,ind.sple[i]], col = cols, axes = F)
  box()
}
```



```
par(mfrow = c(1,1))
```

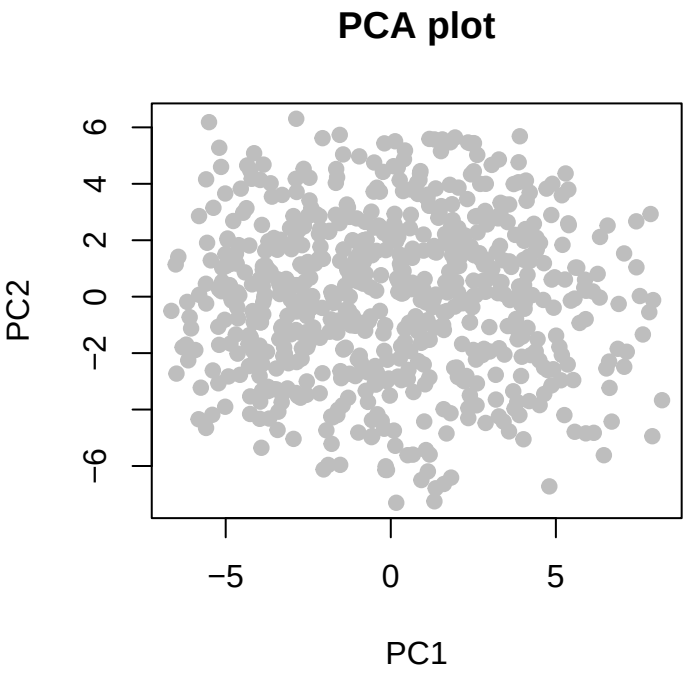
3.2 Question 2

Pour pouvoir effectuer une ACP il nous faut une matrice de dimension nObservations x nVariables. Il nous faut donc convertir chacune des vignettes en vecteur.

```
X = apply(I, 3, function(x){as.vector(x)})
X = t(X)
```

3.3 Question 3

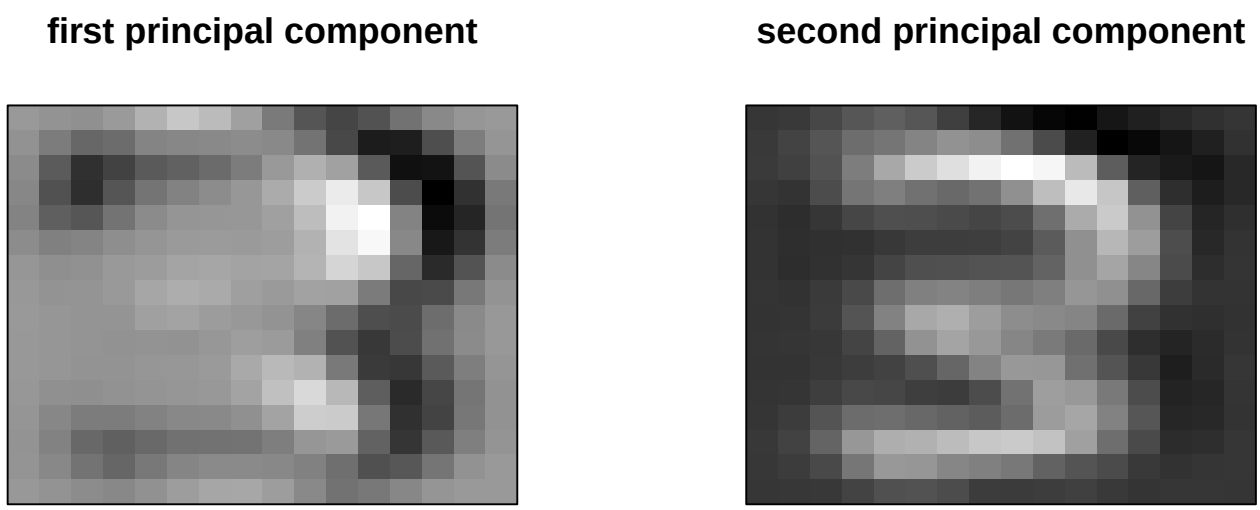
```
pca = prcomp(X)
plot(pca$x[,1], pca$x[,2], xlab = "PC1", ylab = "PC2", main = "PCA plot", pch = 19, col = "grey")
```



3.4 Question 4

Les coefficients définissant les composantes principales sont stockés dans les colonnes du champ `rotation` de l'objet renvoyé par `prcomp`. Pour interpréter ces coefficients comme des images, il faut les re-convertir en matrice et les afficher avec le même code couleur que précédemment.

```
I1  = matrix(pca$rotation[,1], nrow = 16)
I2  = matrix(pca$rotation[,2], nrow = 16)
par(mfrow = c(1,2))
image(I1, col = cols, main = "first principal component", axes = F)
box()
image(I2, col = cols, main  = "second principal component", axes = F)
box()
```




```
par(mfrow = c(1,1))
```

La première composante semble contenir deux empreintes de 3 : une plutôt claire au centre, et une sombre, plus grande et présentant un branche supérieure marquée. Un 3 plutôt resséré “activera”” principalement les

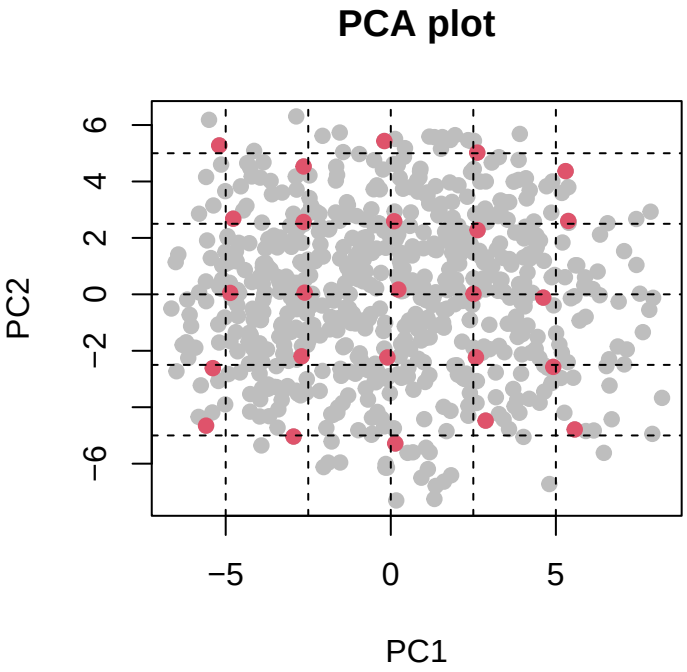
pixels blancs (de poids faibles), et aura une valeur faible sur le 1er axe. A l'inverse, un 3 plutôt grand avec une branche supérieure marquée activera les pixels noirs (de poids fort) et aura une valeur forte sur le 1er axe.

La seconde composante définit une empreinte de 3 homogène de poids négatifs. Plus un caractére sera épais, plus il va "activer" de pixels de cette empreinte (de poids négatifs) et donc avoir une valeur faible sur le 2nd axe.

3.5 Question 5

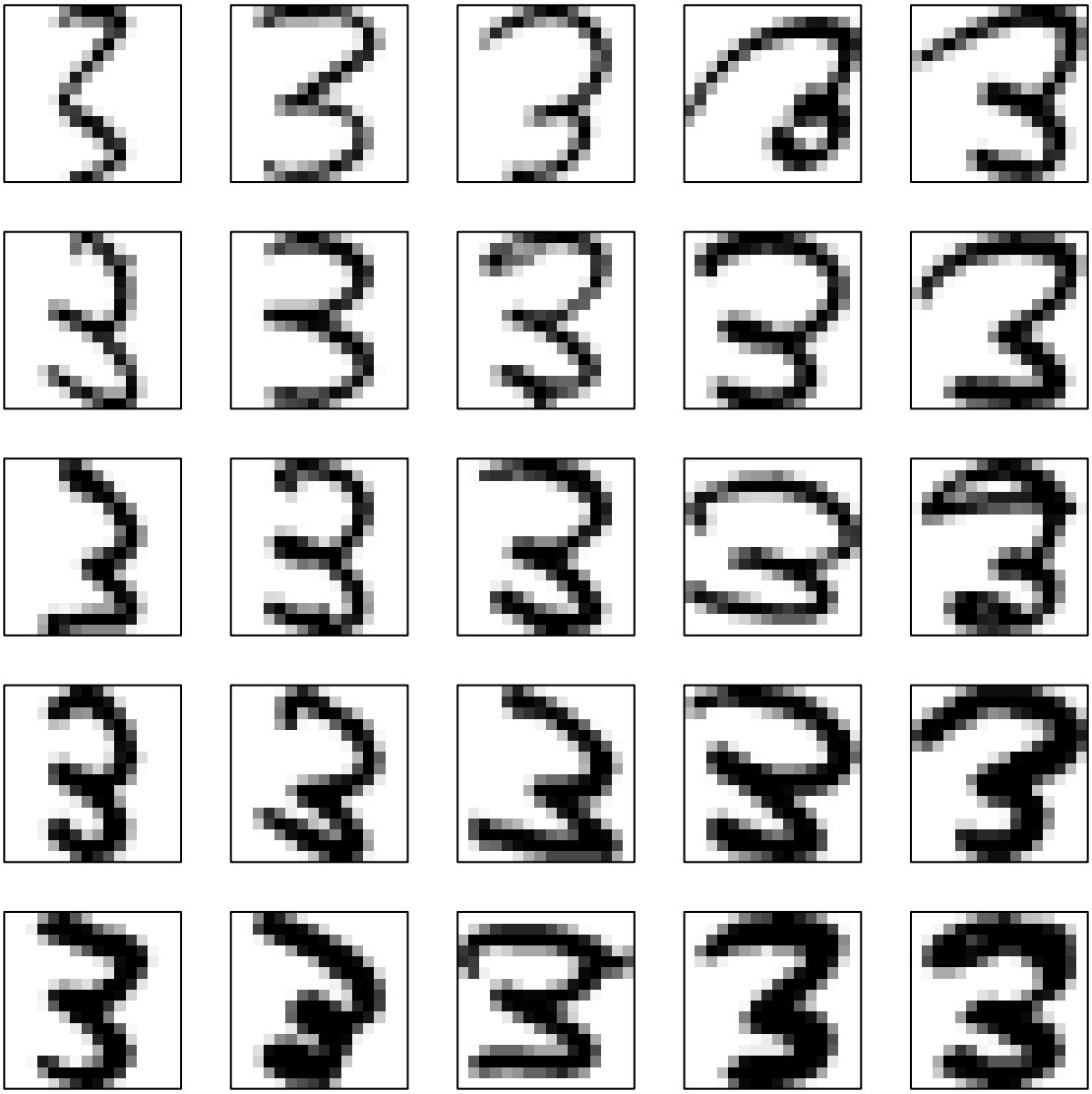
On commence par visualise où se situent les observations répertoriées dans la matrice `ind.grid`. On note qu'elles sont réparties de manière régulière dans cet espace ACP.

```
plot(pca$x[,1], pca$x[,2], xlab = "PC1", ylab = "PC2", main = "PCA plot", pch = 19, col = "grey")
# show points
points(pca$x[ind.grid,1], pca$x[ind.grid,2], col = 2, pch = 19)
abline(h = c(-5,-2.5,0,2.5,5), lty = 2)
abline(v = c(-5,-2.5,0,2.5,5), lty = 2)
```



En regardant comment les images correspondantes évoluent de gauche à droite et de bas en haut, on peut confirmer l'interprétation des composantes principales.

```
# show corresponding images
par(mfrow = c(5,5))
par(mar = c(1,1,1,1))
for(i in 1:5){
  for(j in 1:5){
    image(I[,,ind.grid[i,j]], col = cols, axes = F)
    box()
  }
}
```

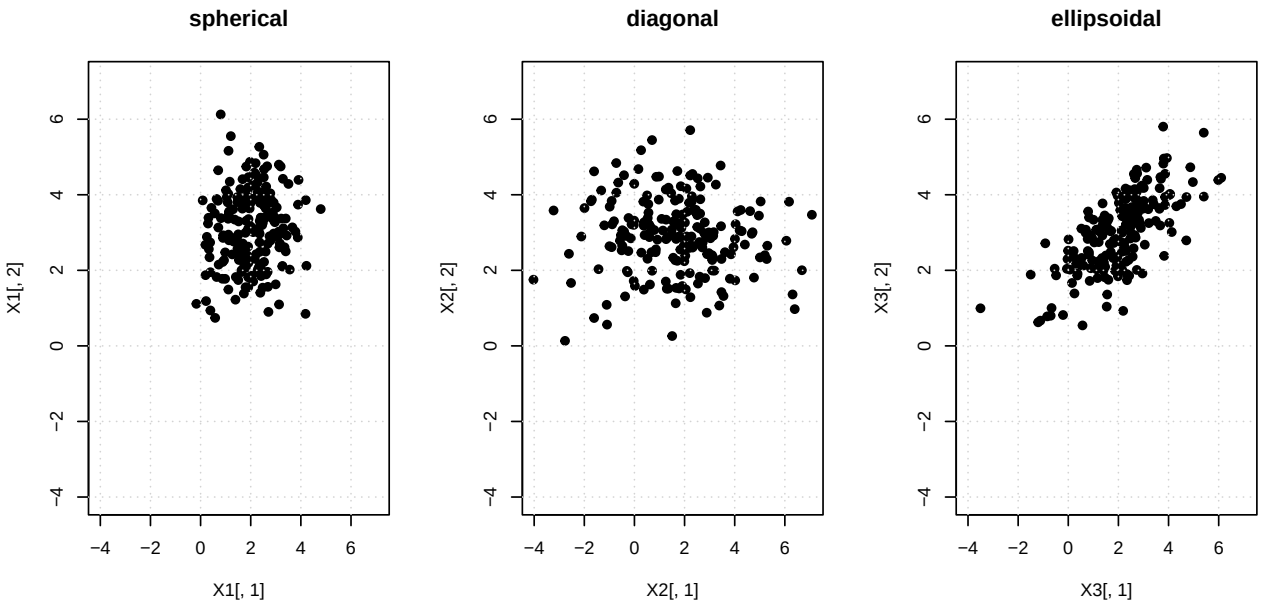


4 Exercice 4

4.1 Question 1

```
############################
#### STARTING EXERCICE 4 ####
############################
library(MASS)
# define mean vector
mu = c(2,3)
# define number of samples to draw
n = 200
# draw - spherical
S1 = matrix(c(1,0,0,1), nrow = 2, byrow = T)
X1 = mvrnorm(n, mu, S1)
# draw = diagonal
S2 = matrix(c(4,0,0,1), nrow = 2, byrow = T)
X2 = mvrnorm(n, mu, S2)
```

```
# draw = ellipsoidal
S3 = matrix(c(2,1,1,1), nrow = 2, byrow = T)
X3 = mvrnorm(n, mu, S3)
# plot
par(mfrow = c(1,3))
xlim = range(c(X1,X2,X3))
plot(X1[,1],X1[,2], xlim = xlim, ylim = xlim, pch = 19, main = "spherical")
grid()
plot(X2[,1],X2[,2], xlim = xlim, ylim = xlim, pch = 19, main = "diagonal")
grid()
plot(X3[,1],X3[,2], xlim = xlim, ylim = xlim, pch = 19, main = "ellipsoidal")
grid()
```




```
par(mfrow = c(1,1))
```

4.2 Question 2

```
library(mclust)
```

```
## Package 'mclust' version 6.0.1
## Type 'citation("mclust")' for citing this R package in publications.
```

```
fit1 = mvn("XII", X1)
cat("*** spherical distribution *** \n")
```

```
## *** spherical distribution ***
```

```
cat("t- estimated mean =", fit1$parameters$mean, "(vs mu =", mu, ")\n")
```

```
## t- estimated mean = 2.019178 3.096562 (vs mu = 2 3 )
```

```
cat("t- estimated Sigma =", fit1$parameters$variance$Sigma, "(vs Sigma =", S1, ")\n")
```

```
## t- estimated Sigma = 0.9562135 0 0 0.9562135 (vs Sigma = 1 0 0 1 )
```

```
fit2 = mvn("XXI", X2)
cat("*** diagonal distribution *** \n")
```

```
## *** diagonal distribution ***
```

```
cat("t- estimated mean =", fit2$parameters$mean, "(vs mu =", mu, ")\n")
```

```
## t- estimated mean = 1.633554 2.906433 (vs mu = 2 3 )
```

```
cat("t- estimated Sigma =", fit2$parameters$variance$Sigma, "(vs Sigma =", S2, ")\n")
```

```
## t- estimated Sigma = 3.910068 0 0 0.9788228 (vs Sigma = 4 0 0 1 )
```

```
fit3 = mvn("XXX", X3)
cat("*** ellipsoidal distribution *** \n")
```

```
## *** ellipsoidal distribution ***
```

```
cat("t- estimated mean =", fit3$parameters$mean, "(vs mu =", mu, ")\n")
```

```
## t- estimated mean = 2.054559 3.023027 (vs mu = 2 3 )
```

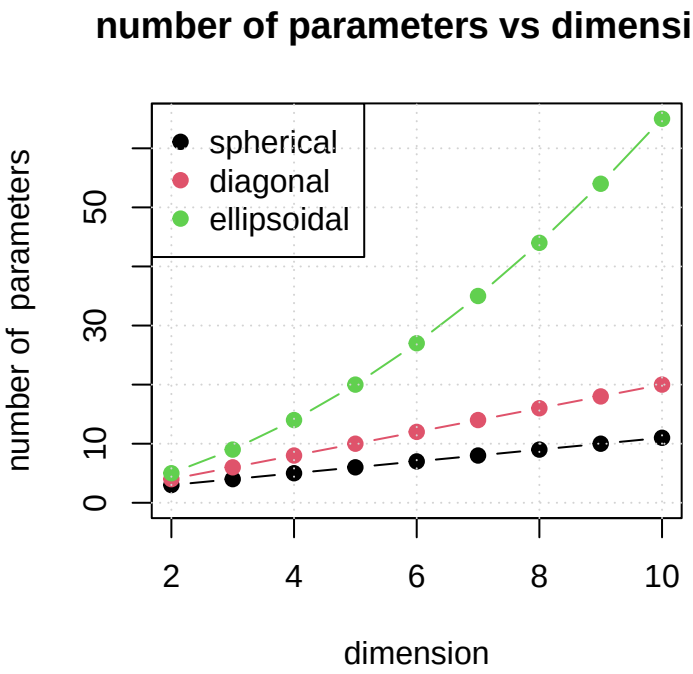
```
cat("t- estimated Sigma =", fit3$parameters$variance$Sigma, "(vs Sigma =", S3, ")\n")
```

```
## t- estimated Sigma = 1.983383 0.9766052 0.9766052 0.9882946 (vs Sigma = 2 1 1 1 )
```

4.3 Question 3

Il faut p paramètres pour estimer le vecteur moyen. La matrice de covariance met en jeu 1 paramètre pour le modèle sphérique, p paramètres pour le modèle diagonal, et $p(p+1)/2$ paramètres pour le modèle général (i.e., ellipsoïdal).

```
# number of dimensions to conisder
p  = seq(2, 10)
# number of parameters of the 3 models
n1 = p + 1
n2 = p + p
n3 = p + p*(p+1)/2
# plot
plot(p, n1, ylim = c(0, max(c(n1,n2,n3))),  type = "b",  pch = 19, xlab = "dimension", ylab = "number of
lines(p, n2, type = "b", pch = 19, col = 2)
lines(p, n3, type = "b", pch = 19, col = 3)
legend("topleft", c("spherical","diagonal","ellipsoidal"), col = c(1,2,3), pch = 19)
grid()
```



4.4 Question 4

On commence par implémenter une fonction calculant la densité de la loi normale multivariée.

```
mv = function(X, mu, S){
 p = length(mu)
 return( 1/( sqrt((2*pi)^p*det(S)) ) *  exp( -0.5 * t(X-mu) %*% solve(S) %*% (X-mu)) )
}
```

On peut visualiser la densité en utilisant la fonction `contour`. Pour cela il suffit de calculer sa valeur sur une grille discrétisant l'espace (x, y). Dans cet exemple nous calculons deux densités : les densités théoriques et estimées du modèle ellipsoïdal précédent.

```
# define grid
x = seq(-2,6, by = 0.02)
y = x
# init density
D = matrix(0, nrow = length(x), ncol = length(y))
D.hat = D
# extract estimated parameters
mu.hat= as.vector(fit3$parameters$mean)
S3.hat = fit3$parameters$variance$Sigma
# compute densities
for(i in 1:nrow(D)){
  for(j in 1:ncol(D)){
    X = matrix( c(x[i],y[j]), nrow = 2, ncol = 1)
    D[i,j] = mv(X, mu, S3)
    D.hat[i,j]  = mv(X, mu.hat, S3.hat)
  }
}
```

Enfin, on trace les deux densités sur le même graphique, avec les points tirés précédemment.

```
# plot
plot(X3[,1], X3[,2], pch = 19, col = "grey")
```

```r
contour(x,y,D, add = TRUE)
contour(x,y,D.hat, add = TRUE, col = "red")
legend("bottomright", c("densité théorique","densité estimée"), col = c("black","red"), lwd = 1)
```