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

SY09 : TP 08 : Analyse discriminante

Julien Jerphanion

Printemps 2018

Table des matières

1 Programmation

1.1 Analyse discriminante

Ici, on va s'intéresser à différentes implémentations des analyses discriminantes quadratiques et linéaire ainsi qu'à celle du classifieur bayésien naïf. `Xapp` et `zapp` représente le jeu de données à apprendre et leurs étiquettes.

```
adq.app = function(Xapp, zapp) {
  #' Apprentissage de l'ADQ
  #'
  #' @param Xapp jeu de données pour l'apprentissage
  #' @param zapp étiquettes du jeu de données d'apprentissage
    n = dim(Xapp)[1]
    p = dim(Xapp)[2]
    g = max(unique(zapp))

    param = NULL
    param$MCov = array(0, c(p,p,g))
    param$mean = array(0, c(g,p))
    param$prop = rep(0, g)

    for (k in 1:g)
    {
        indk = which(zapp==k)
        X_k = X[indk,]

        param$MCov[,,k] = cov.wt(X_k)$cov
        param$mean[k,] = colMeans(X_k)
        param$prop[k] = length(indk) / n
    }

    param
}
```

```
adl.app = function(Xapp, zapp) {
  #' Apprentissage de l'ADL
  #'
  #' @param Xapp jeu de données pour l'apprentissage
  #' @param zapp étiquettes du jeu de données d'apprentissage
    n = dim(Xapp)[1]
    p = dim(Xapp)[2]
    g = max(unique(zapp))

    param = NULL
    MCov = array(0, c(p,p))
    param$MCov = array(0, c(p,p,g))
    param$mean = array(0, c(g,p))
    param$prop = rep(0, g)

    for (k in 1:g)
    {
        indk = which(zapp==k)
        X_k = X[indk,]
```

```
        param$mean[k,] = colMeans(X_k)
        param$prop[k] = length(indk) / n
        MCov = MCov + param$prop[k] * cov.wt(X_k)$cov

    }
    for (k in 1:g)
    {
        param$MCov[,,k] = MCov
    }

    param
}
```

```
nba.app = function(Xapp, zapp) {
  #' Apprentissage du classifieur naïf Bayésien
  #'
  #' @param Xapp jeu de données pour l'apprentissage
  #' @param zapp étiquettes du jeu de données d'apprentissage
    n = dim(Xapp)[1]
    p = dim(Xapp)[2]
    g = max(unique(zapp))

    param = NULL
    param$MCov = array(0, c(p,p,g))
    param$mean = array(0, c(g,p))
    param$prop = rep(0, g)

    for (k in 1:g)
    {
        indk = which(zapp==k)
        X_k = X[indk,]

        param$MCov[,,k] = diag(diag(cov.wt(X_k)$cov))
        param$mean[k,] = colMeans(X_k)
        param$prop[k] = length(indk) / n
    }

    param
}
```

```
ad.val = function(param, Xtst) {
  #' Calcule les probabilités a posteriori pour un ensemble de données,
  #' puis effectue le classement en fonction de ces probabilités
  #'
  #' @param param paramètres du modèle
  #' @param Xtst paramètres
    n = dim(Xtst)[1]
    p = dim(Xtst)[2]
    g = length(param$prop)

    out = NULL

    prob = matrix(0, nrow=n, ncol=g)
```

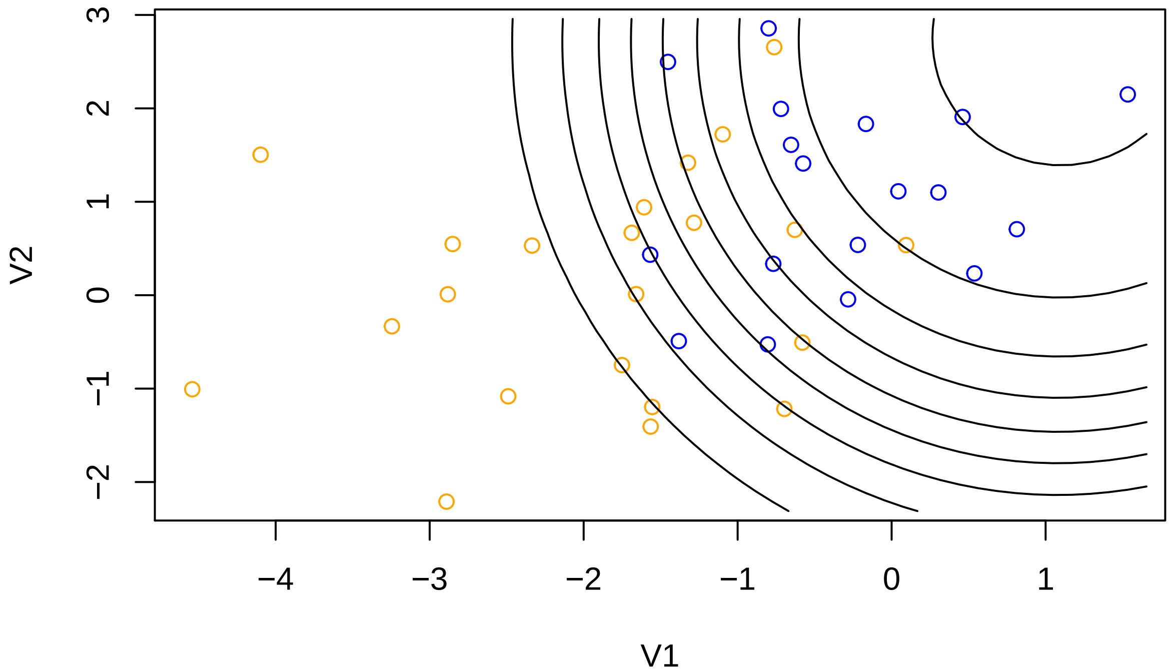
```
    for (k in 1:g)
    {
      mu_k = param$mean[k,]
      MCov_k = param$MCov[,,k]
      prop_k = param$prop[k]
        prob[,k] = prop_k * mvdnorm(Xtst, mu_k, MCov_k)
    }

    pred = max.col(prob)
    prob = prob / rowSums(prob)

    out$prob = prob
    out$pred = pred

    out
}
```

```
# Importation de dépendances
source("fonctions/mvdnorm.r")
source("fonctions/prob.ad.R")
```

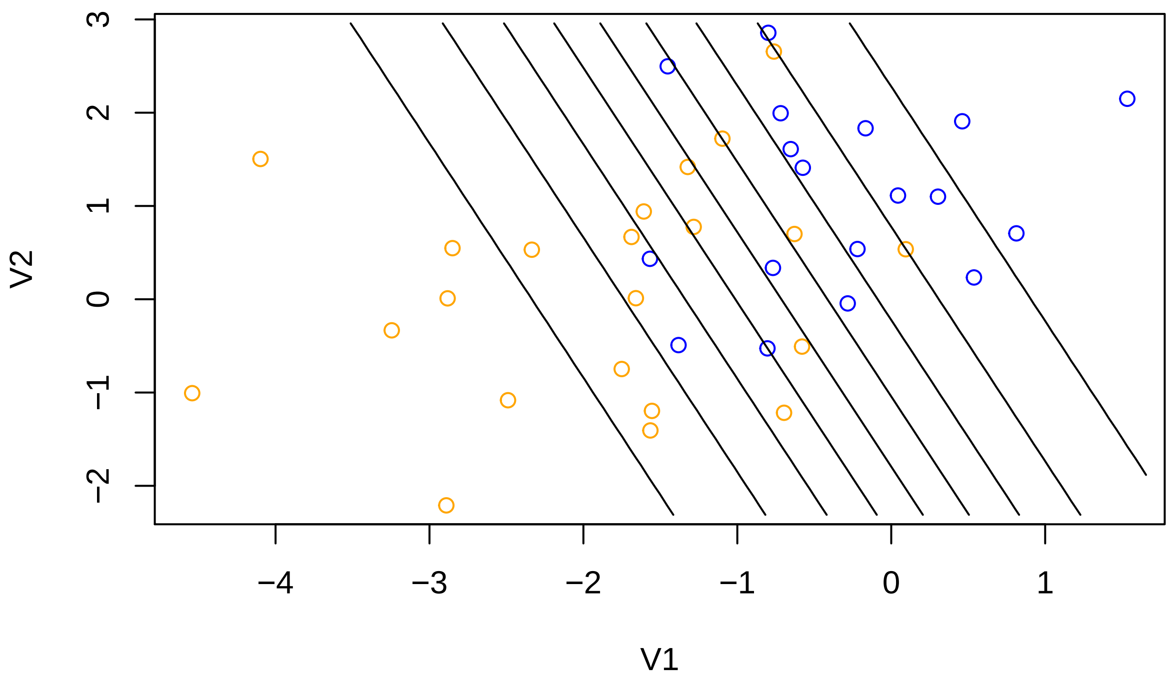
1.2 Vérification des fonctions

```
data = read.csv("../TP06/donnees/Synth1-40.csv")
X = data[,1:2]
z = data[,3]

# Apprentissage sur tout le jeu de données
adq.params = adq.app(Xapp = X, zapp = z)
adl.params = adl.app(Xapp = X, zapp = z)
nba.params = nba.app(Xapp = X, zapp = z)
```

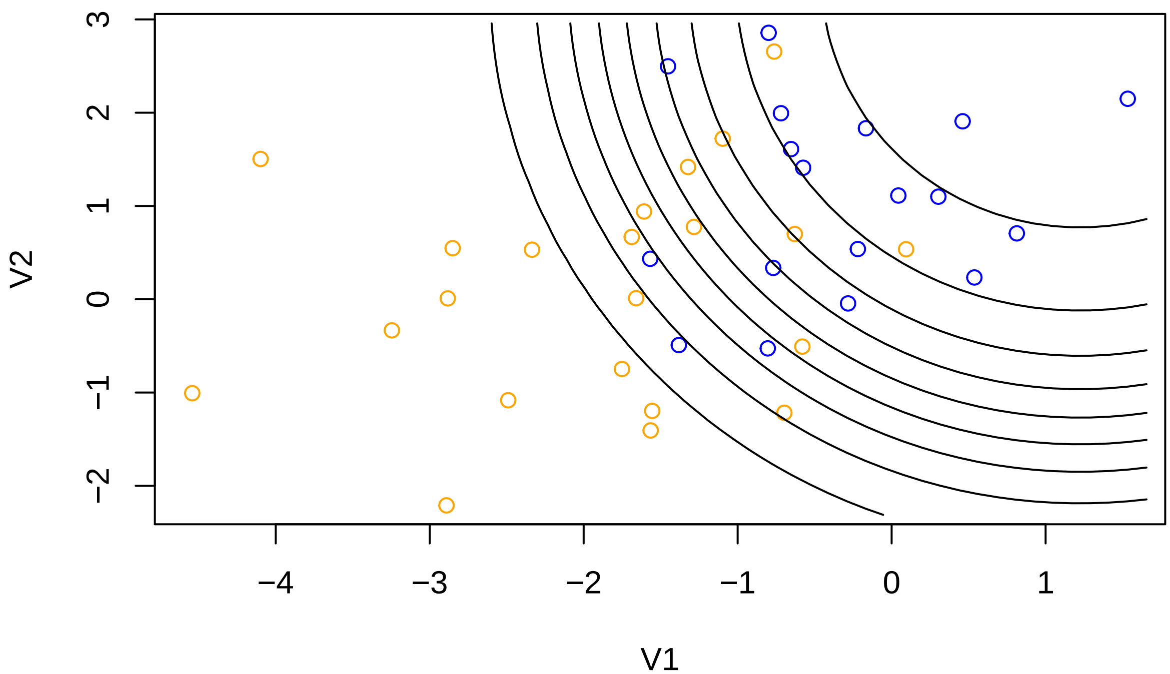
```
# Affichage des frontière de décisions
niv = 1:10 / 10.
prob.ad(param = adq.params, X = X, z = z,niveaux = niv)
```




```r
prob.ad(param = adl.params, X = X, z = z,niveaux = niv)
```




```
prob.ad(param = nba.params, X = X, z = z,niveaux = niv)
```



On retrouve bien les résultats escomptés pour les frontières, à savoir:

1. un ellipsoïde pour l'analyse discriminante quadratique ;
2. une variété linéaire pour l'analyse discriminante linéaire ;
3. une ellispoïde dont les axes sont portés par la base canonique pour le classifieur bayésien linéaire ;

2 Applications

Appliquons cela à des jeux de données. Avant de ce faire, on charge quelques dépendances pour la séparation des jeux de données.

```
source("../TP06/fonctions/separ1.R")
source("../TP06/fonctions/separ2.R")
```

2.1 Test sur données simulées

On met en place la fonction suivante pour l'évaluation des modèles.

```
ad.models = function(X,z, model, niter=20, test_size = 1/3) {
  #' Détermination des erreurs pour un model donné
  #'
  #' @param X : jeu de données
  #' @param z : les étiquettes du jeu de données
  #' @param model : modèle à utiliser
```

```
  #' @param niter : le nombre d'estimation à réaliser
  #' @param test_size : proportion à considérer pour l'ensemble de test

  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 = separ1(X,z,test_size)
    Xapp = donn$Xapp
    Xtst = donn$Xtst
    zapp = donn$zapp
    ztst = donn$ztst

    params = model(Xapp,zapp)

    zappClass = ad.val(params,Xapp)$pred
    ztstClass = ad.val(params,Xtst)$pred

    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
}
```

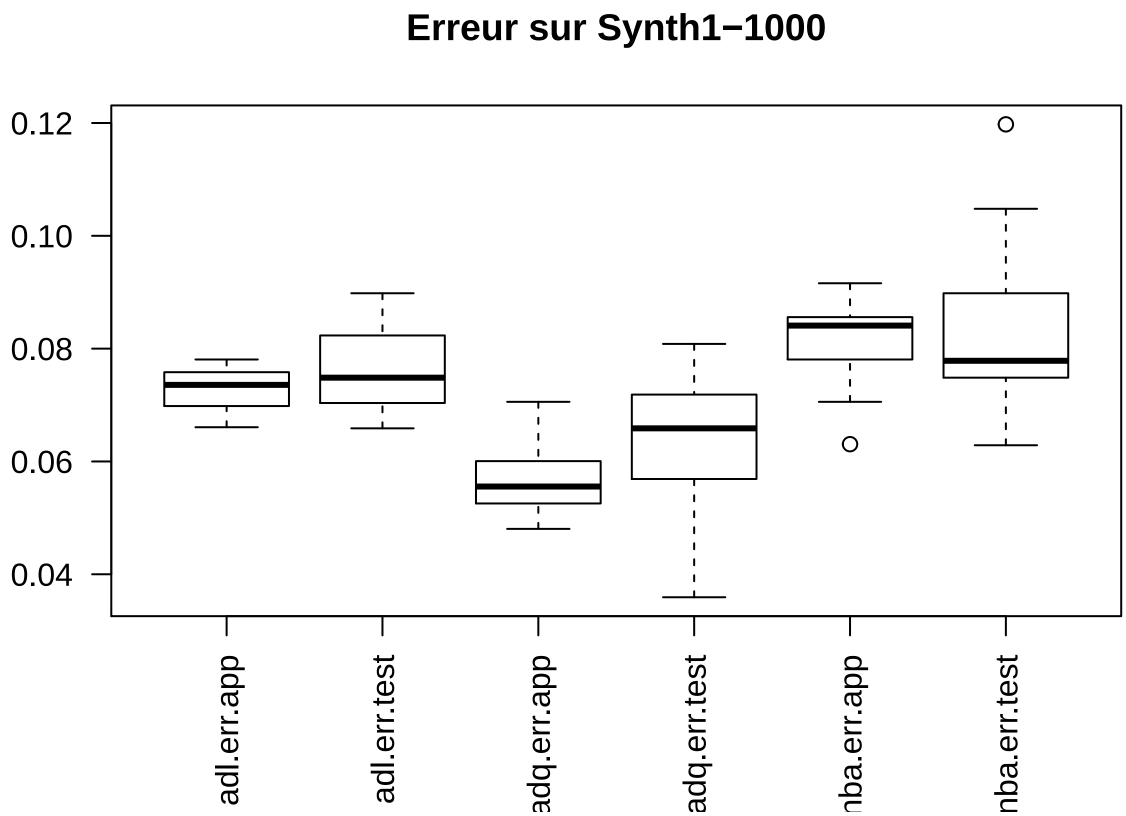
On définit aussi de la plomberie pour charger les jeux de données :

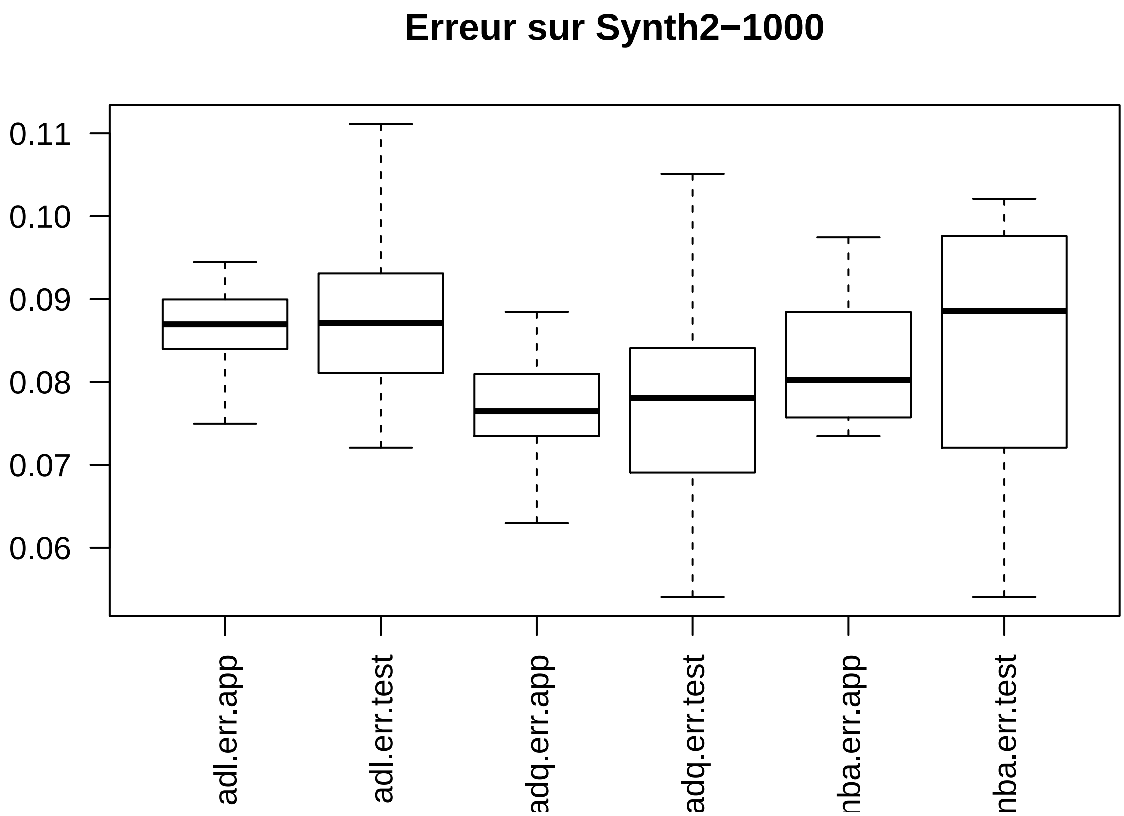
```
dataSets = c("Synth1-1000","Synth2-1000","Synth3-1000")

loadData = function(dataSet) {
  #' Charge un jeu de données de deux dimensions
  dataSetName = paste("./donnees/",dataSet,".csv",sep = "")
  donn = read.csv(dataSetName)
  p = dim(donn)[2] -1
  X = donn[,1:p]
  z = donn[,p+1]
```

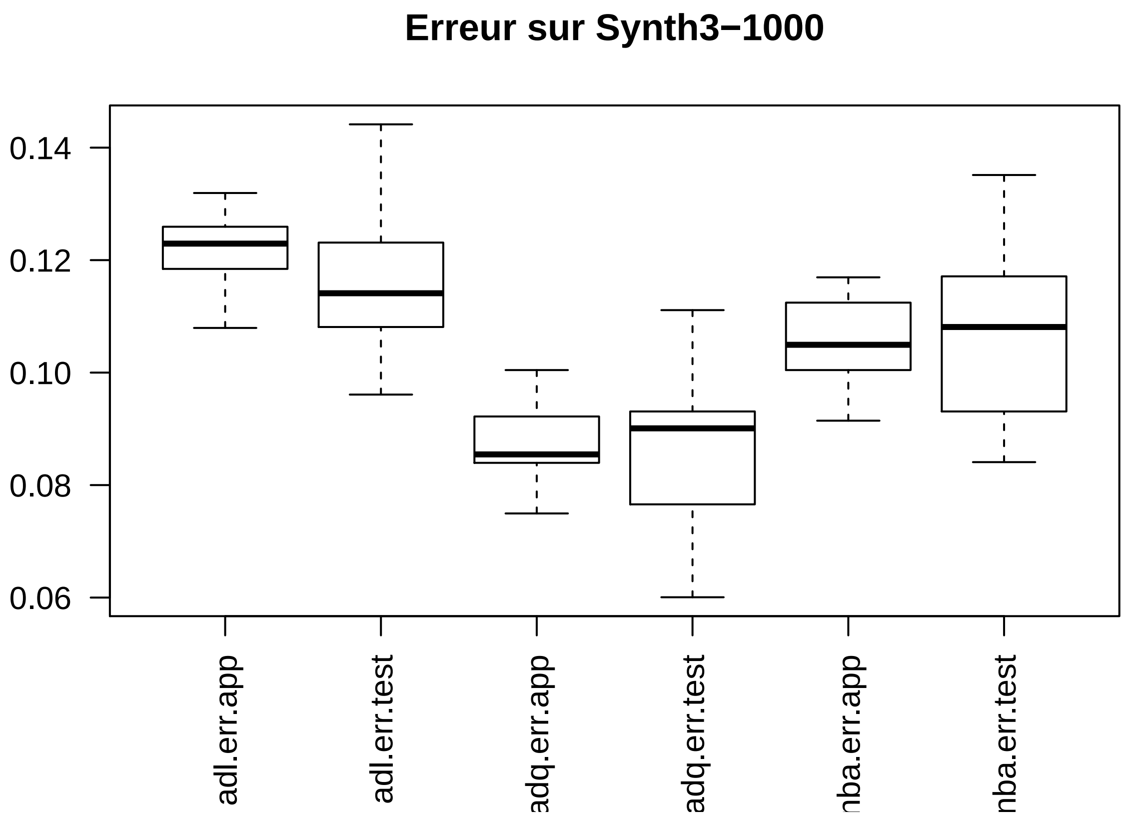
```
  data = NULL
  data$X = X
  data$z = z

  data
}
```

On peut alors facilement obtenir les erreurs d'apprentissage et de test selon les jeux de données et les méthodes.

```
for(i in 1:length(dataSets)) {
  dataSet = dataSets[i]
  data = loadData(dataSet)
  X = data$X
  z = data$z

  # Apprentissage avec (2/3,1/3)
  adl.res = ad.models(X,z,adl.app)
  adq.res = ad.models(X,z,adq.app)
  nba.res = ad.models(X,z,nba.app)

  list.err = list(adl.err.app = adl.res$errApp,
                  adl.err.test = adl.res$errTest,
                  adq.err.app = adq.res$errApp,
                  adq.err.test = adq.res$errTest,
                  nba.err.app = nba.res$errApp,
                  nba.err.test = nba.res$errTest)

  boxplot(list.err,main = paste("Erreur sur",dataSet), las=2)
}
```

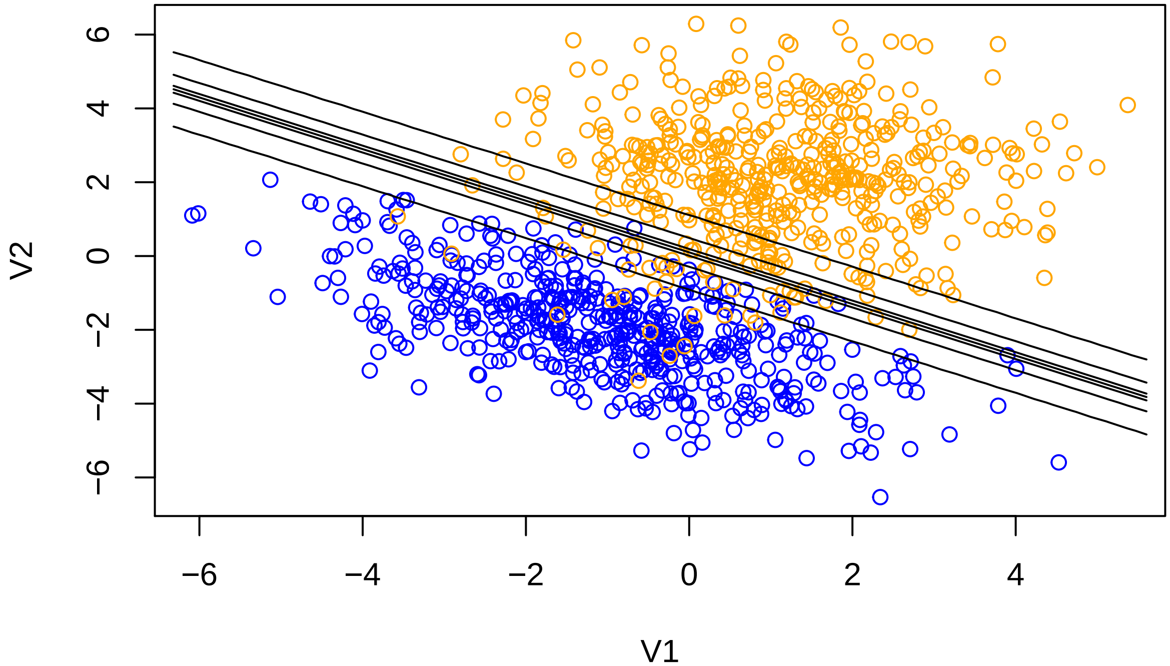

Erreur sur Synth1−1000
0.12
0.10
0.08
0.06
0.04
adl.err.app
adl.err.test
adq.err.app
adq.err.test
nba.err.app
nba.err.test

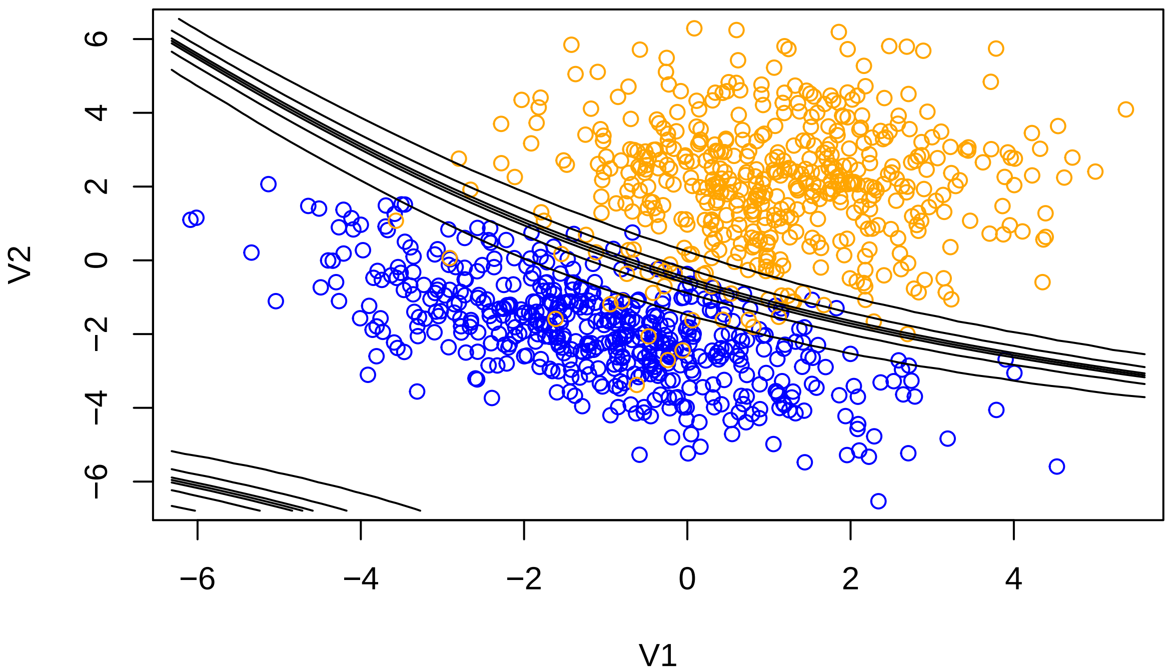

Erreur sur Synth2−1000
0.11
0.10
0.09
0.08
0.07
0.06
adl.err.app
adl.err.test
adq.err.app
adq.err.test
nba.err.app
nba.err.test

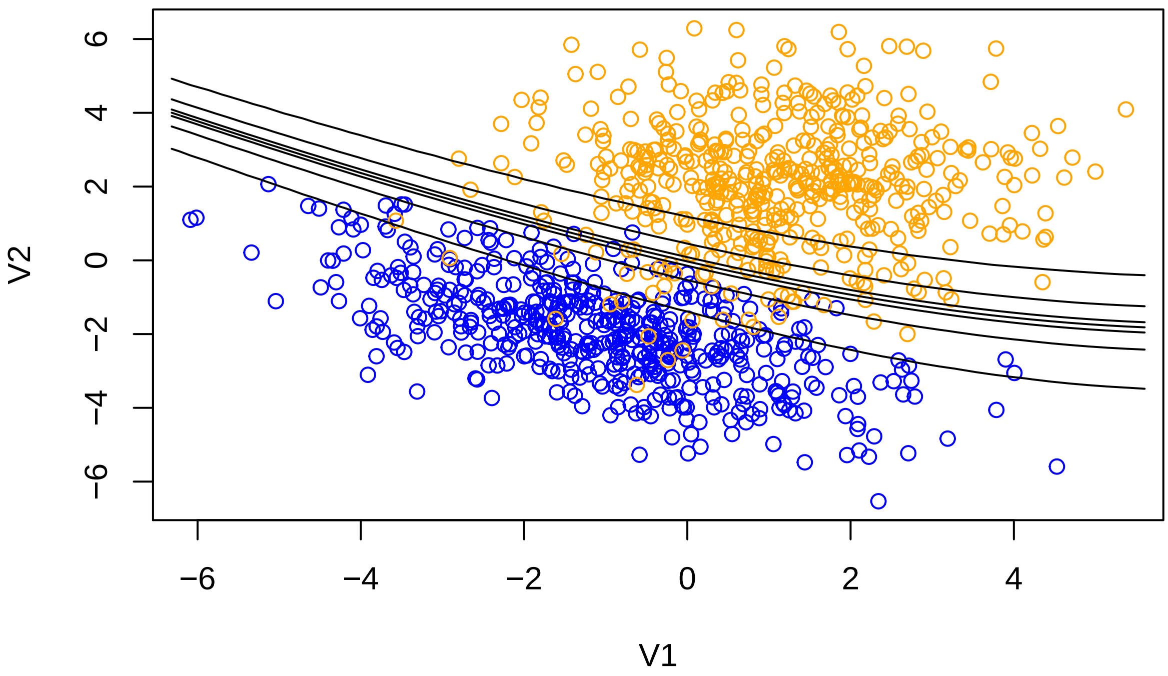


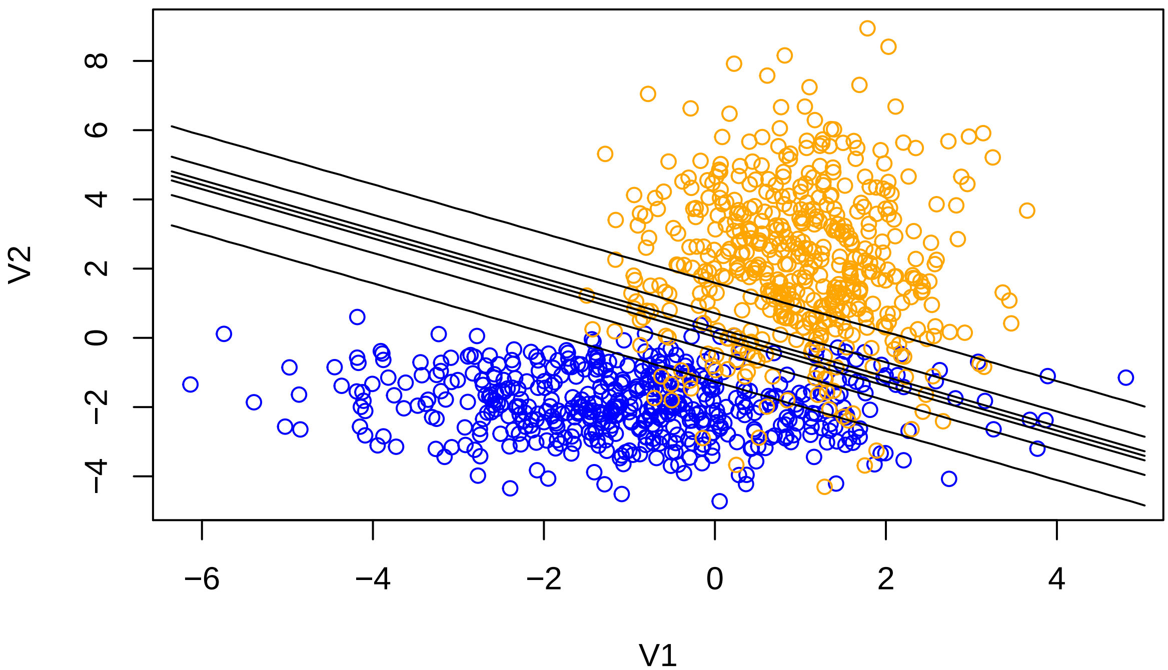
On peut s'intéresser à la frontière de décisions dans ces cas :

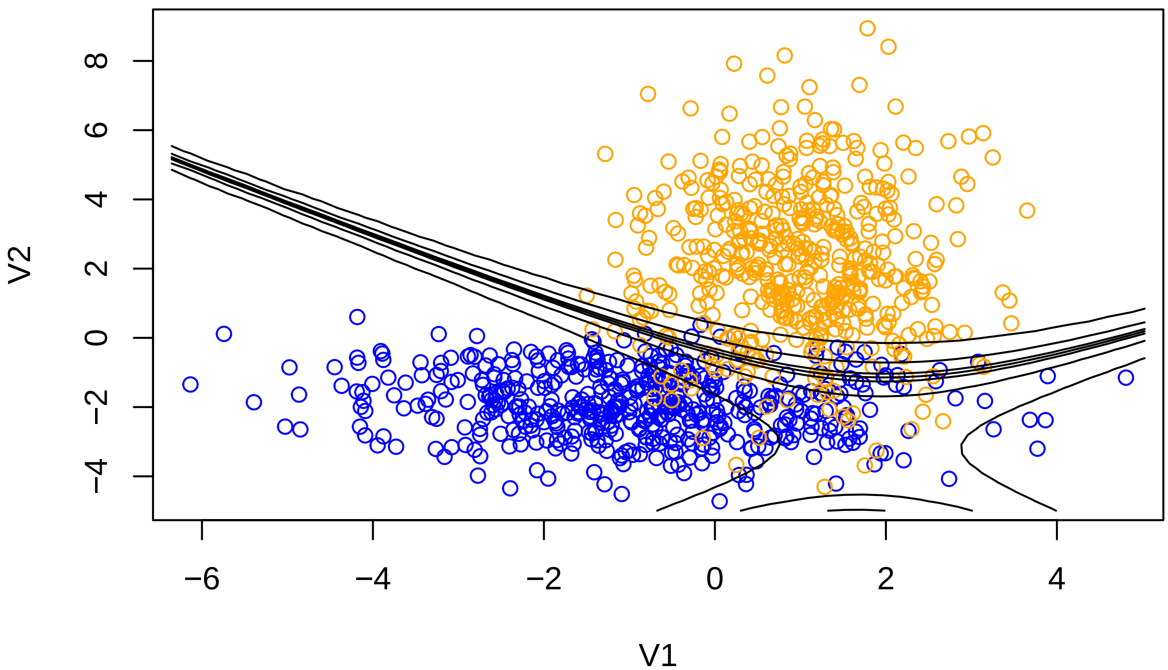
```
for(i in 1:length(dataSets)) {
  dataSet = dataSets[i]
  data = loadData(dataSet)
  X = data$X
  z = data$z
  levels = c(0.1, 0.3, 0.45,0.5, 0.55,0.7, 0.9)
  adl.params = adl.app(X, z)
  adq.params = adq.app(X, z)
  nba.params = nba.app(X, z)
  prob.ad(adl.params, X,z, levels)
  prob.ad(adq.params, X,z, levels)
  prob.ad(nba.params, X,z, levels)
}
```


V1
V2
−6
−4
−2
0
2
4
6

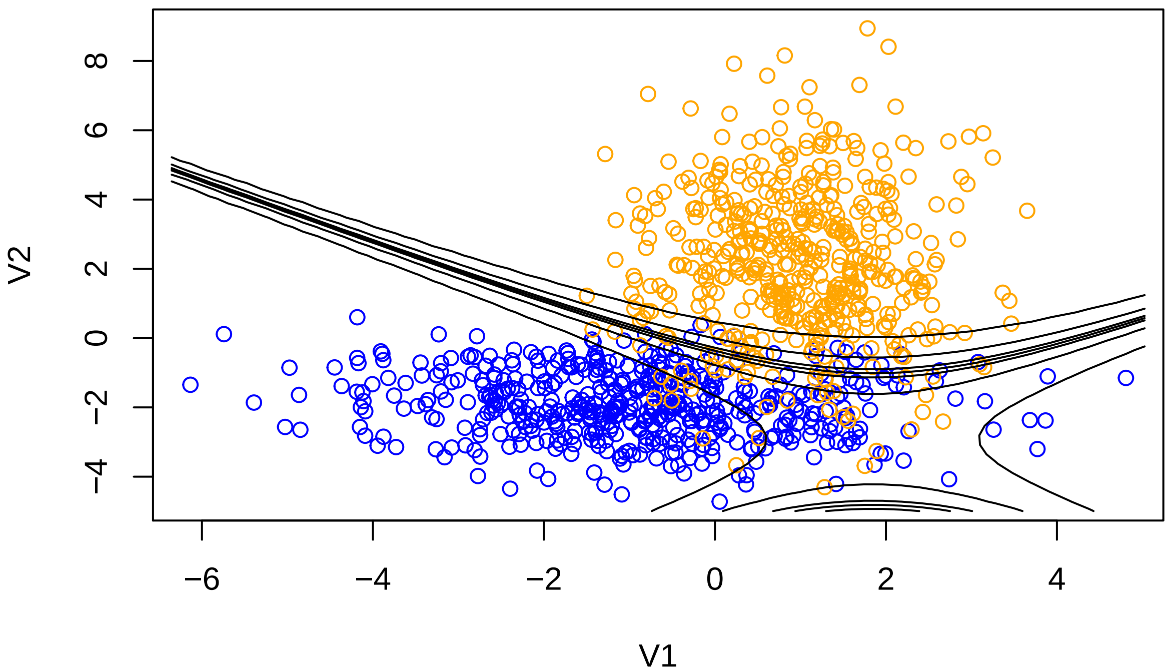

V2
V1
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4

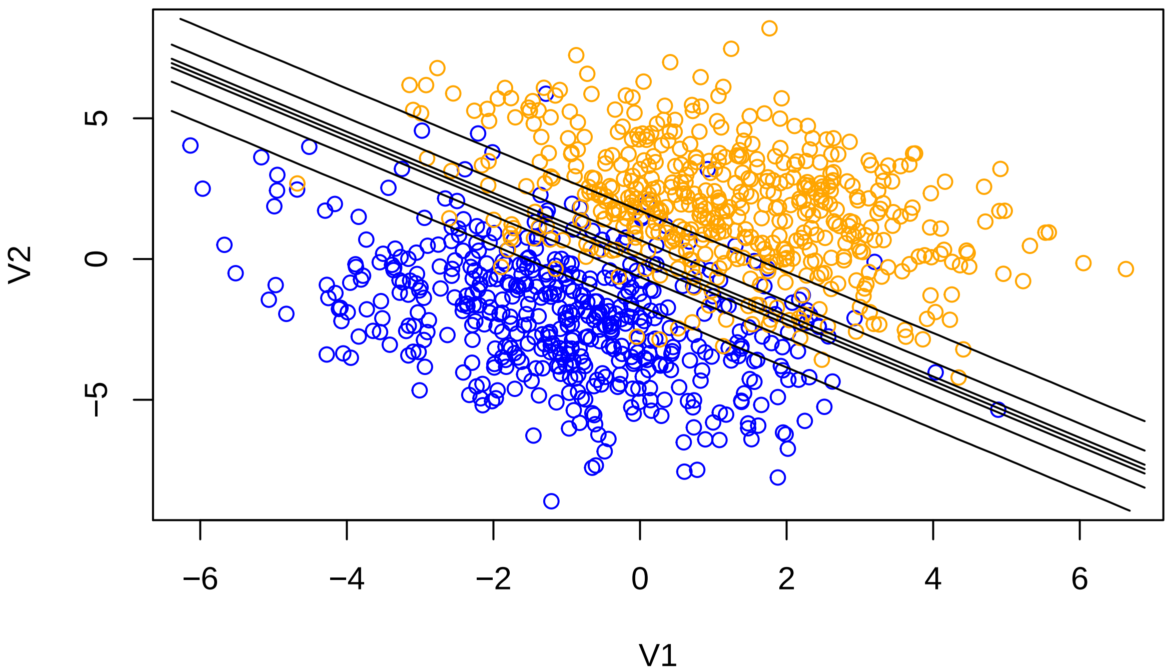

V2
V1
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4

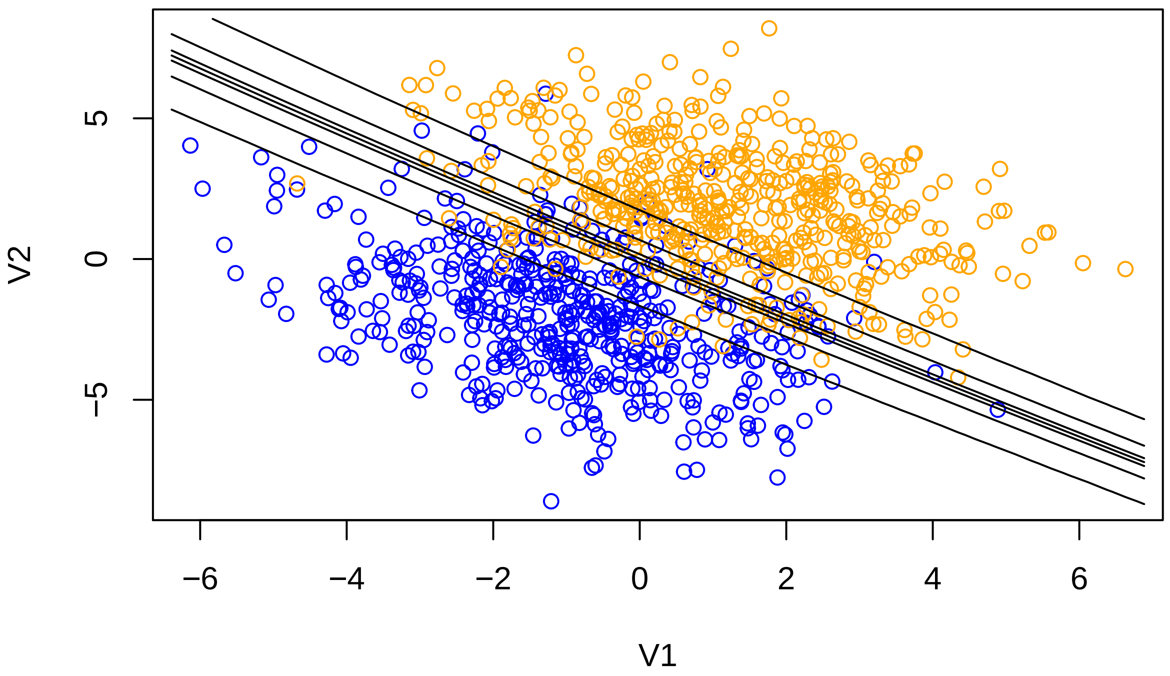

V2
V1
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4

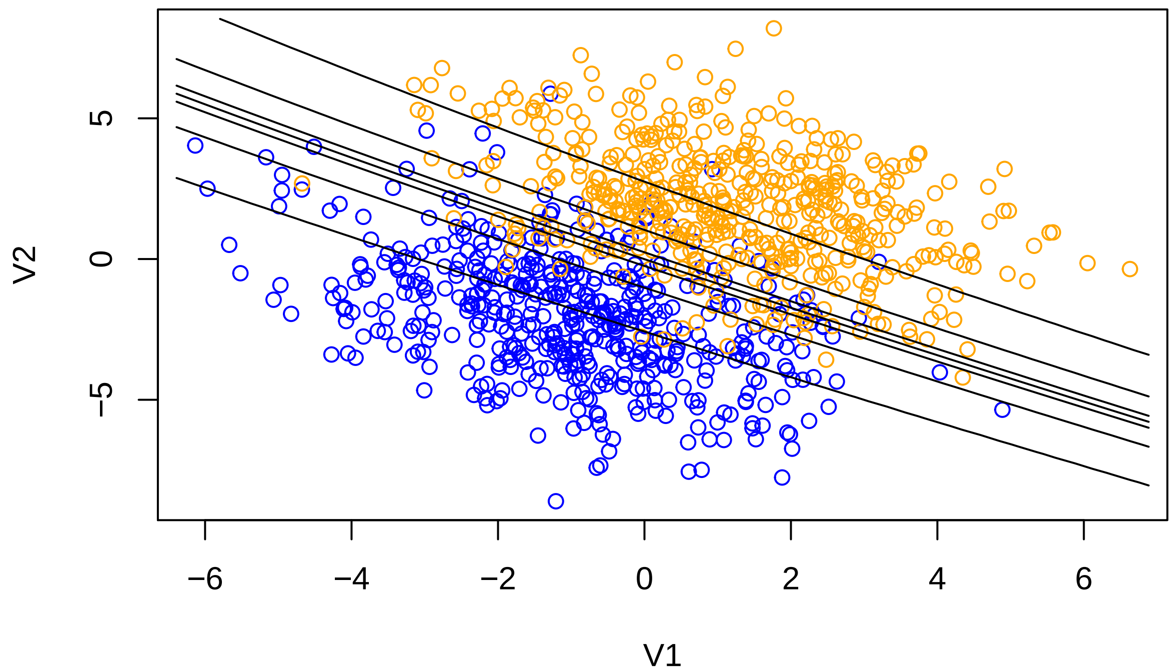


V2
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
V1


V2
V1
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4

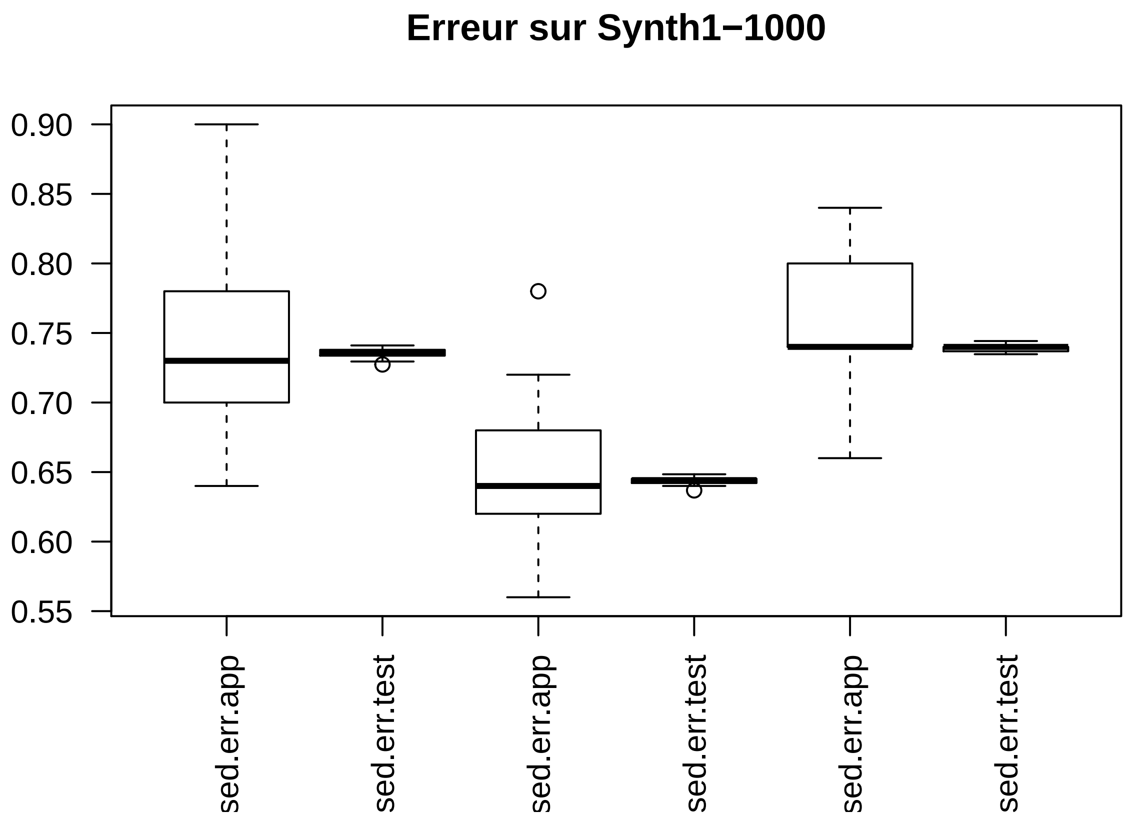

V1
V2
−6
−4
−2
0
2
4
6
5
0
−5

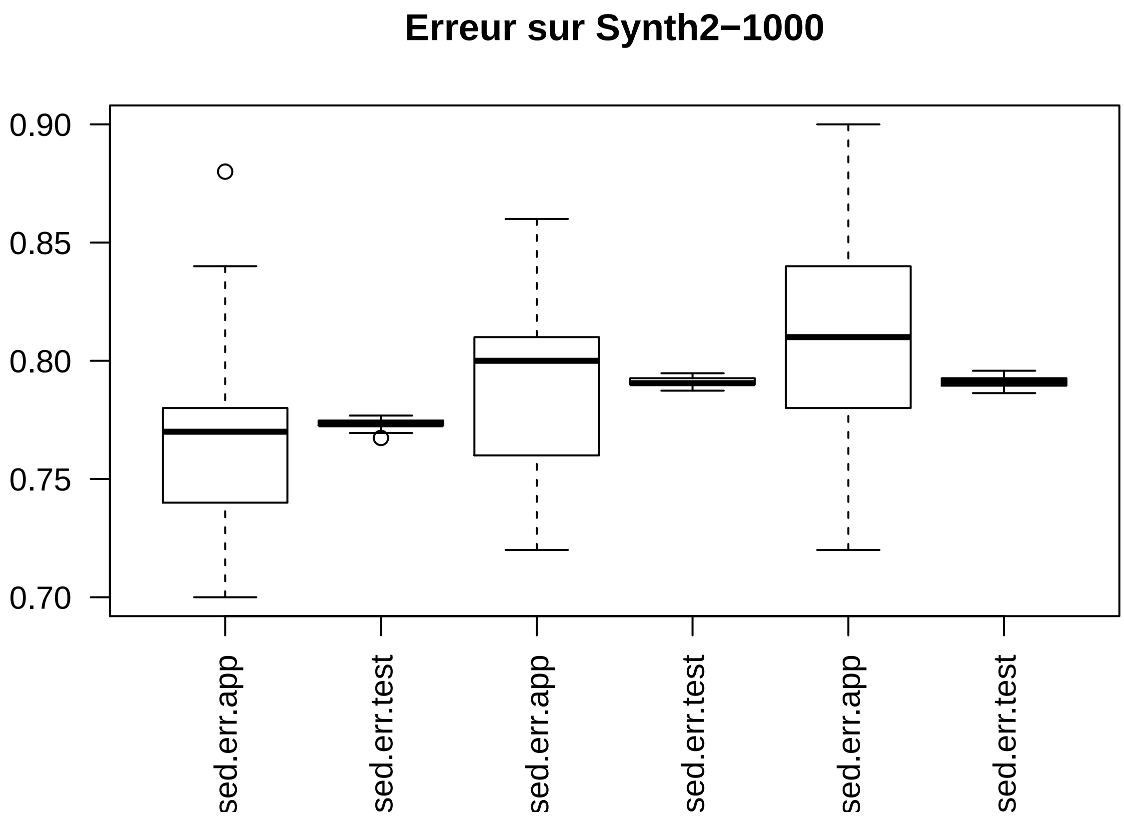

V1
V2
−6
−4
−2
0
2
4
6
−5
0
5

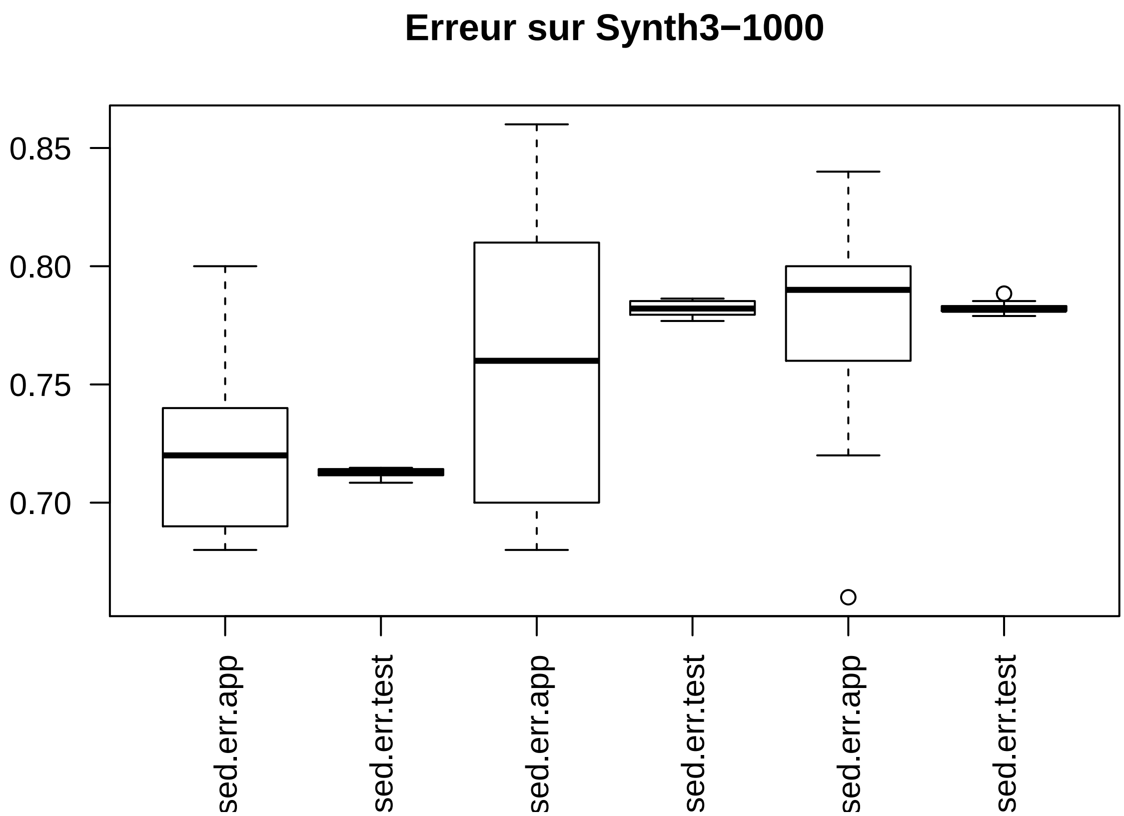


Si on réalise l'apprentissage avec très peu de données (ici avec $n_{app} = 50$ données), on se rend compte que l'erreur augmente de manière significative.

```
for(i in 1:length(dataSets)) {
  dataSet = dataSets[i]
  data = loadData(dataSet)
  X = data$X
  z = data$z

  # Apprentissage avec n_app = 50
  test_size = 1. - 50 / length(z)
  adl.res.biased = ad.models(X,z,adl.app, test_size = test_size)
  adq.res.biased = ad.models(X,z,adq.app, test_size = test_size)
  nba.res.biased = ad.models(X,z,nba.app, test_size = test_size)

  list.err.biased = list(adl.biased.err.app = adl.res.biased$errApp,
                         adl.biased.err.test = adl.res.biased$errTest,
                         adq.biased.err.app = adq.res.biased$errApp,
                         adq.biased.err.test = adq.res.biased$errTest,
                         nba.biased.err.app = nba.res.biased$errApp,
                         nba.biased.err.test = nba.res.biased$errTest)

  boxplot(list.err.biased, main = paste("Erreur sur",dataSet), las=2)
}
```

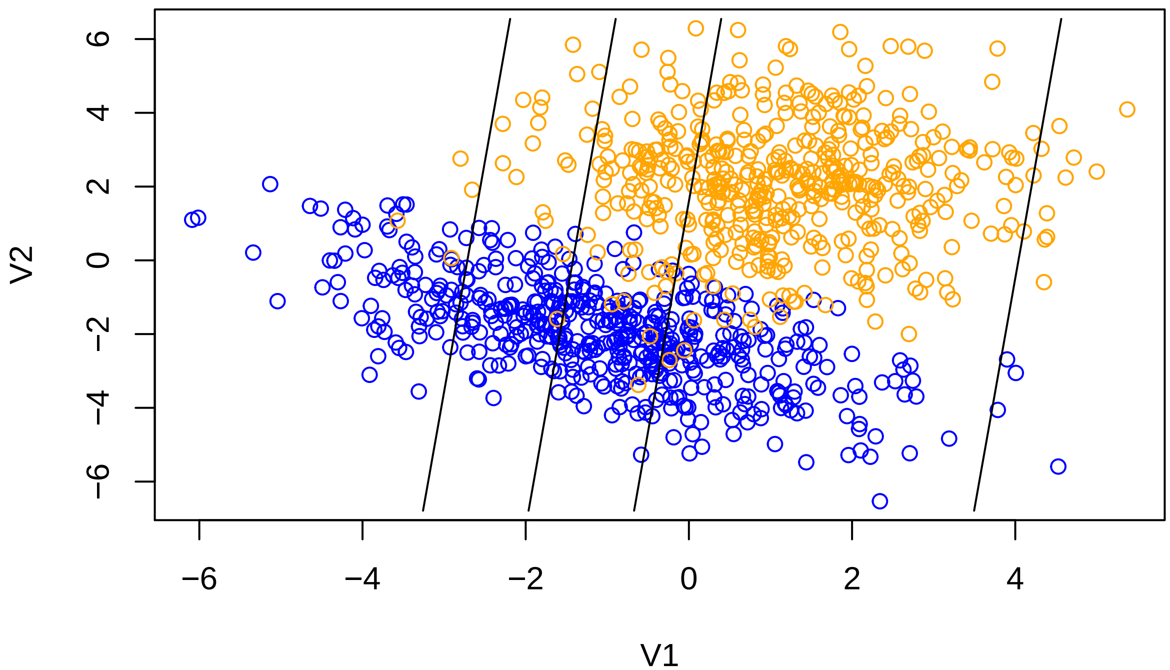

Erreur sur Synth1−1000
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
sed.err.app
sed.err.test
sed.err.app
sed.err.test
sed.err.app
sed.err.test

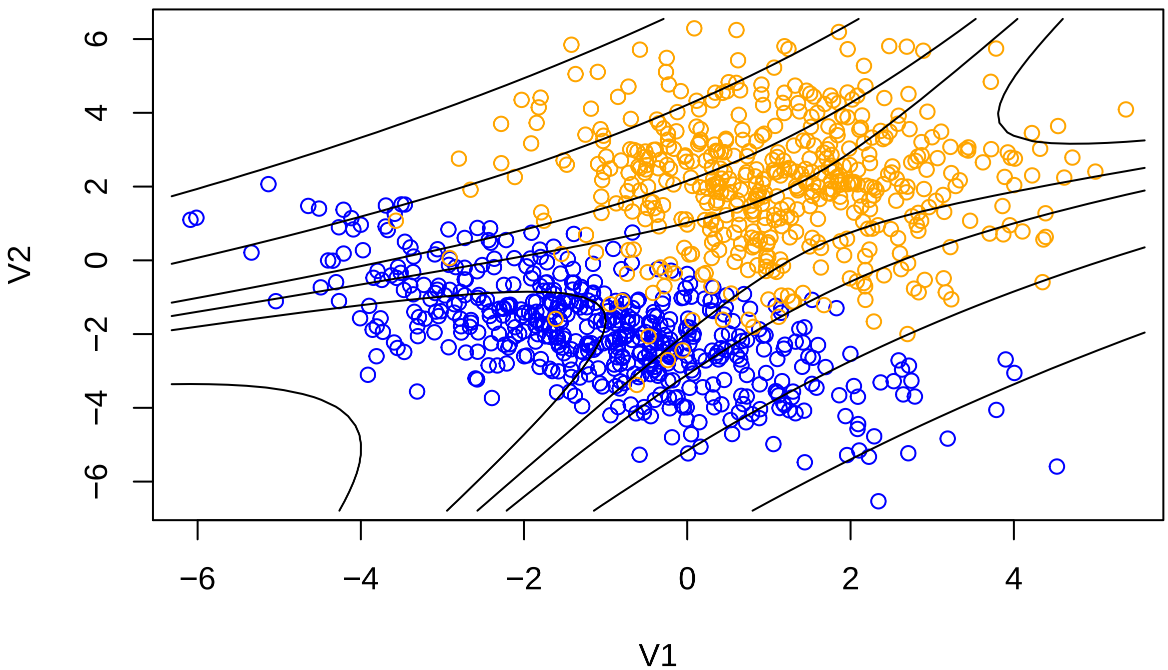

Erreur sur Synth2−1000
0.90
0.85
0.80
0.75
0.70
sed.err.app
sed.err.test
sed.err.app
sed.err.test
sed.err.app
sed.err.test

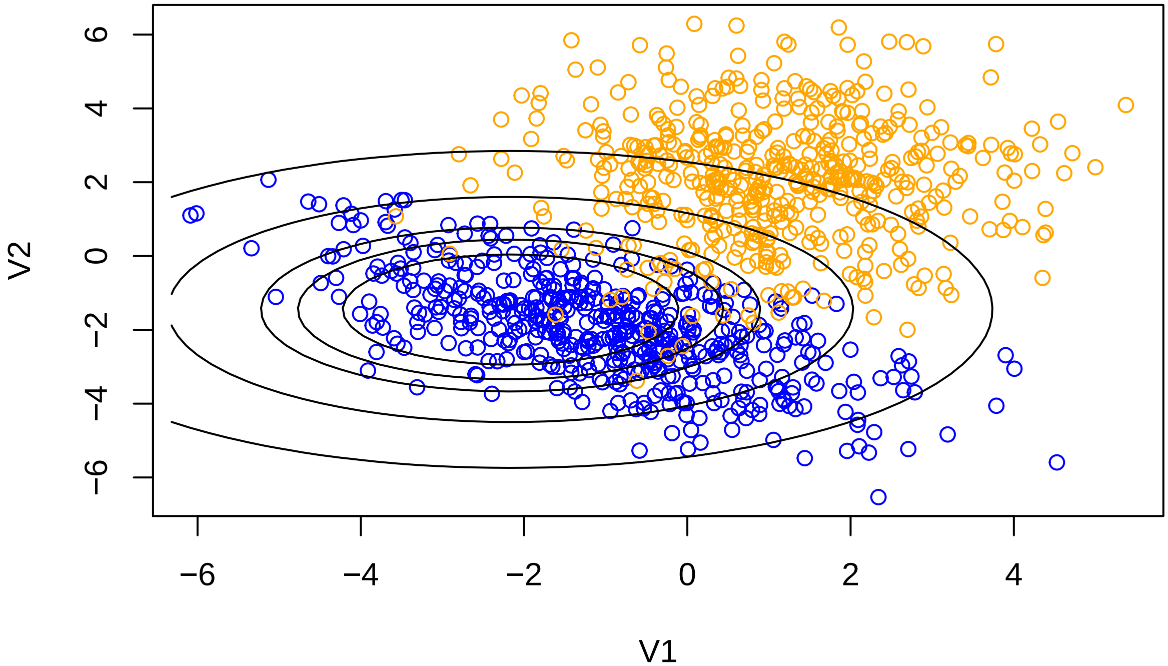


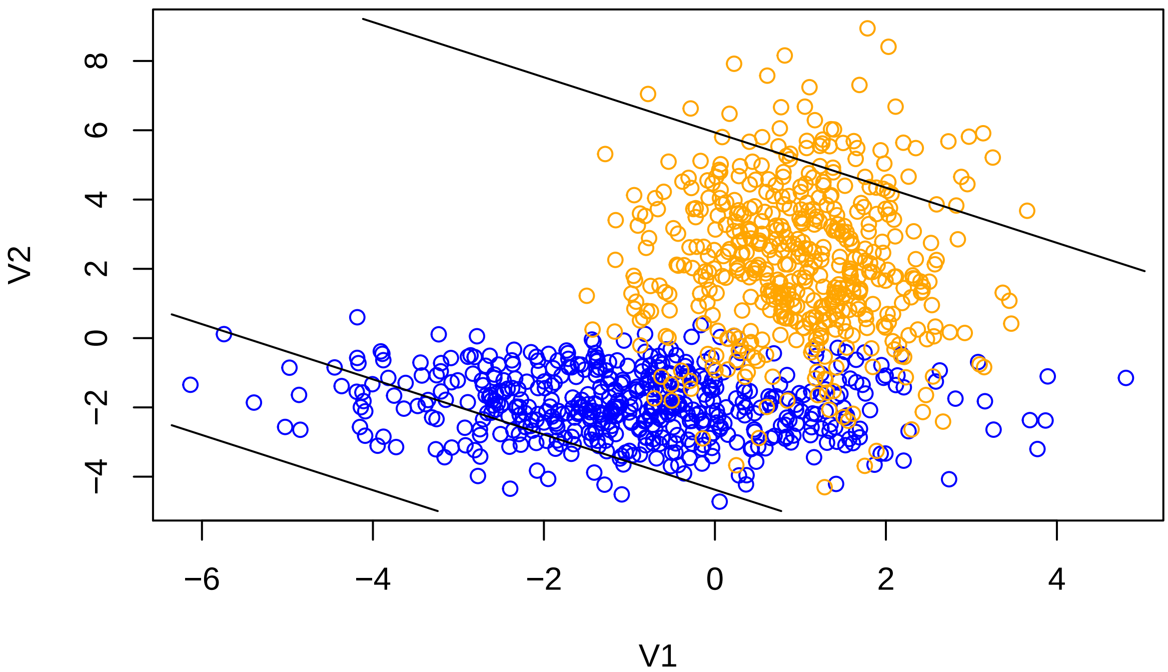
Les frontières de décisions sont ici complètement à l'Ouest.

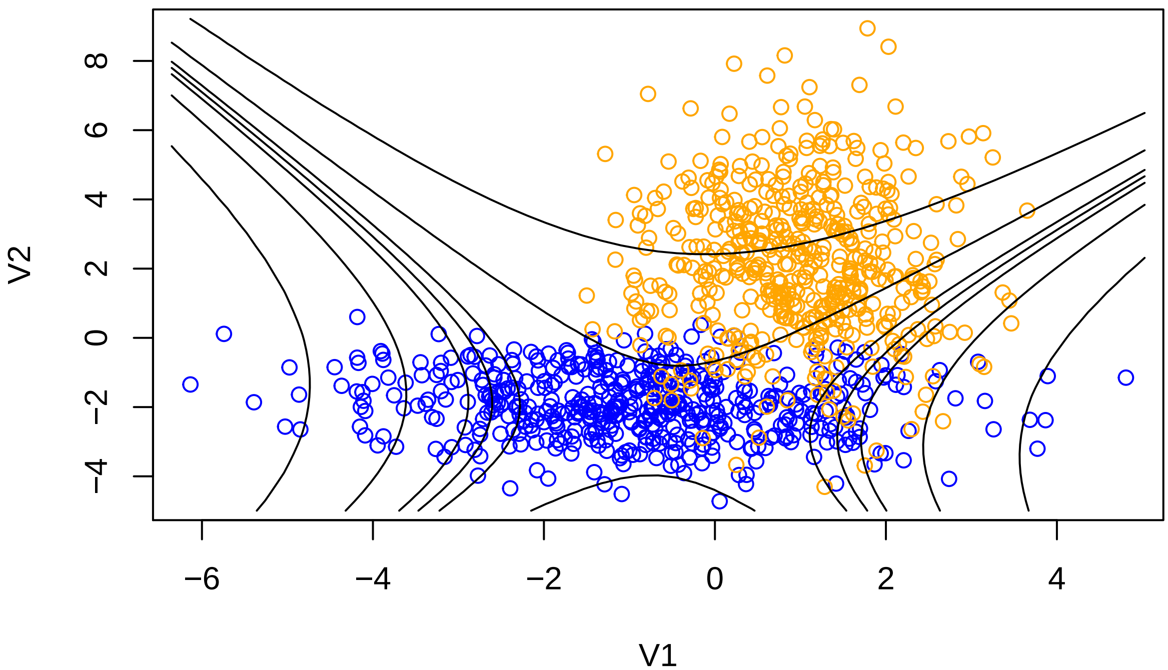
```
for(i in 1:length(dataSets)) {
  dataSet = dataSets[i]
  data = loadData(dataSet)
  X = data$X
  z = data$z
  levels = c(0.1, 0.3, 0.45,0.5, 0.55,0.7, 0.9)
  napp = sample(1:dim(X)[1],50,replace = FALSE)
  Xnapp = X[napp,]
  znapp = z[napp]
  adl.params = adl.app(Xnapp, znapp)
  adq.params = adq.app(Xnapp, znapp)
  nba.params = nba.app(Xnapp, znapp)
  prob.ad(adl.params, X,z, levels)
  prob.ad(adq.params, X,z, levels)
  prob.ad(nba.params, X,z, levels)
}
```

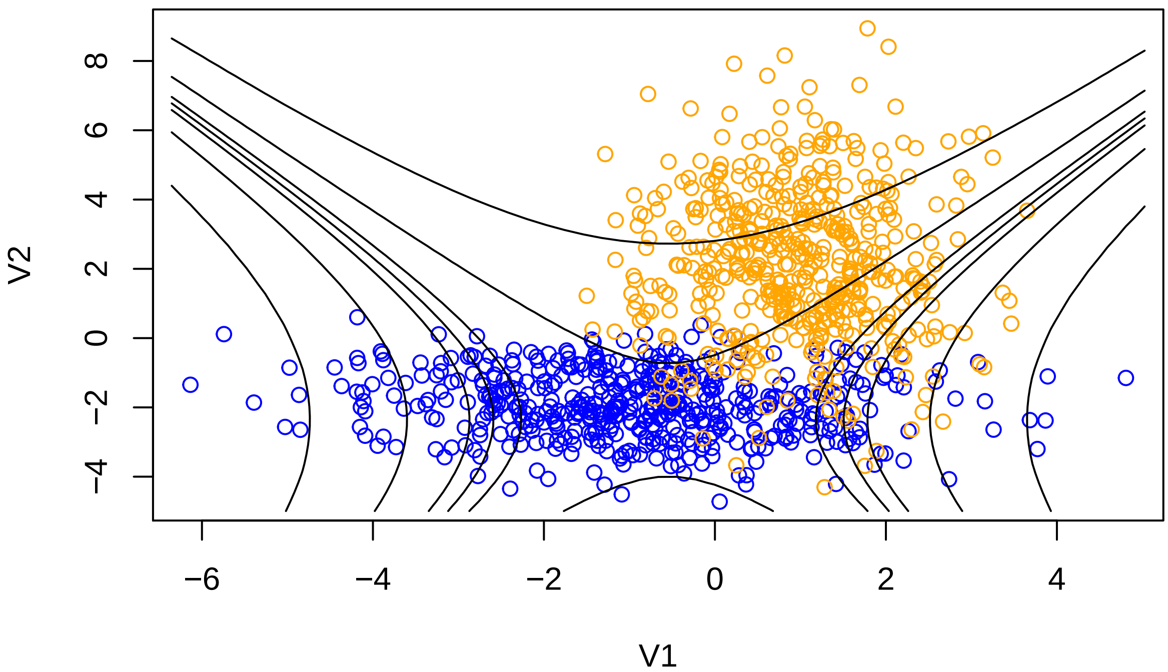

V2
V1
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4

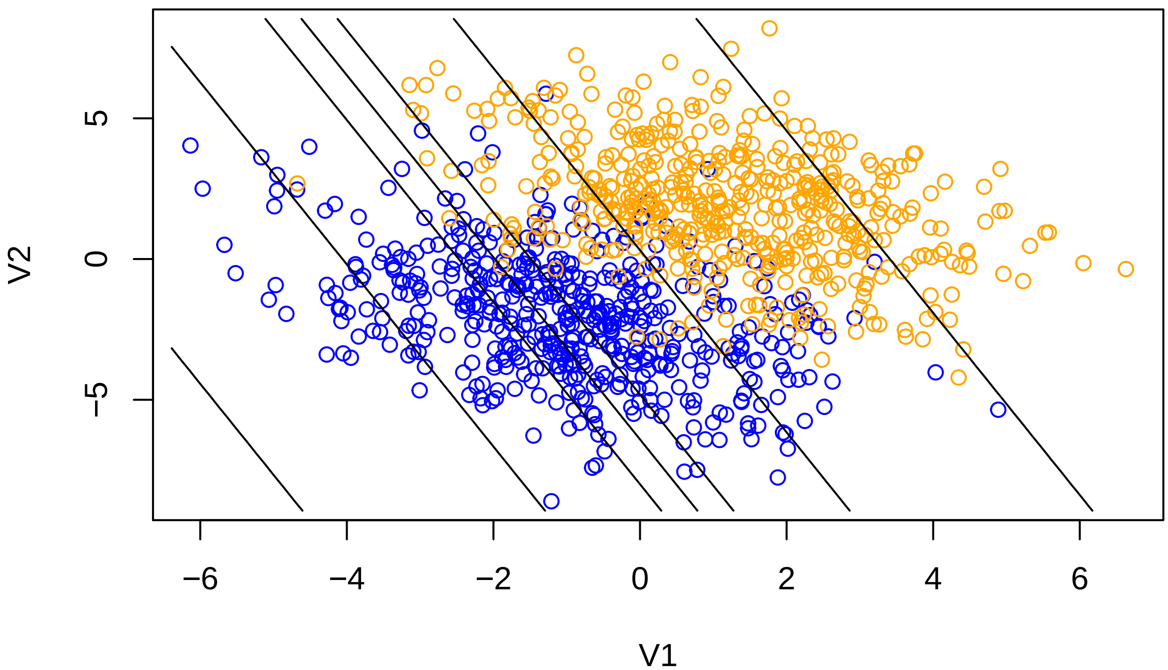

V2
V1
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4

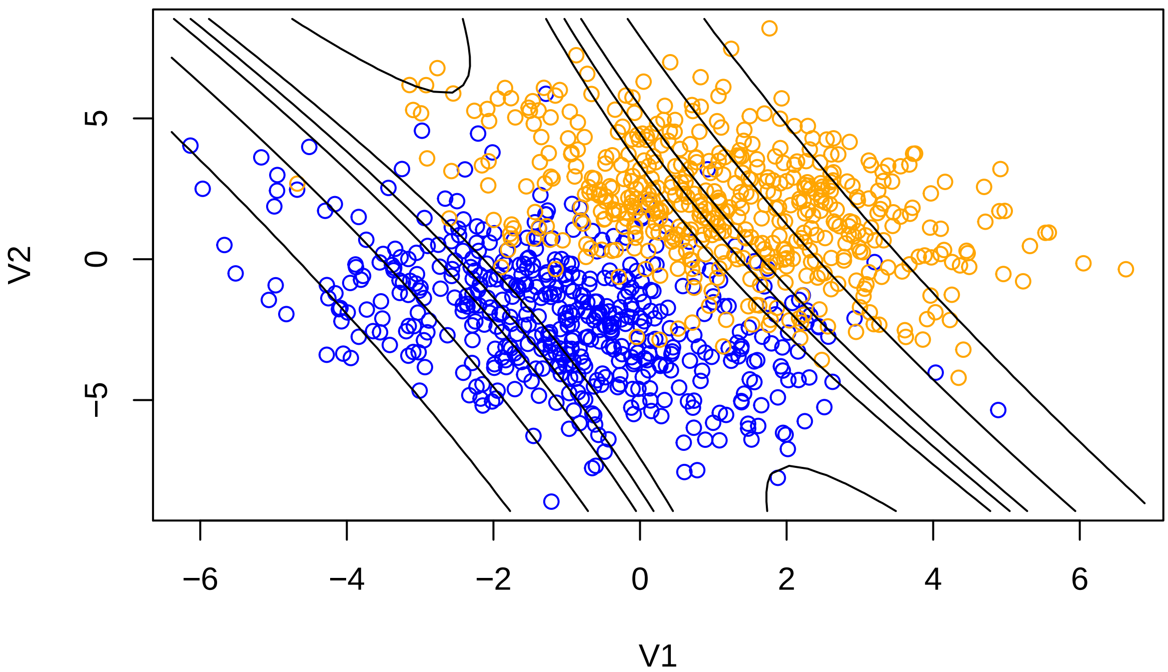

V2
V1
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4

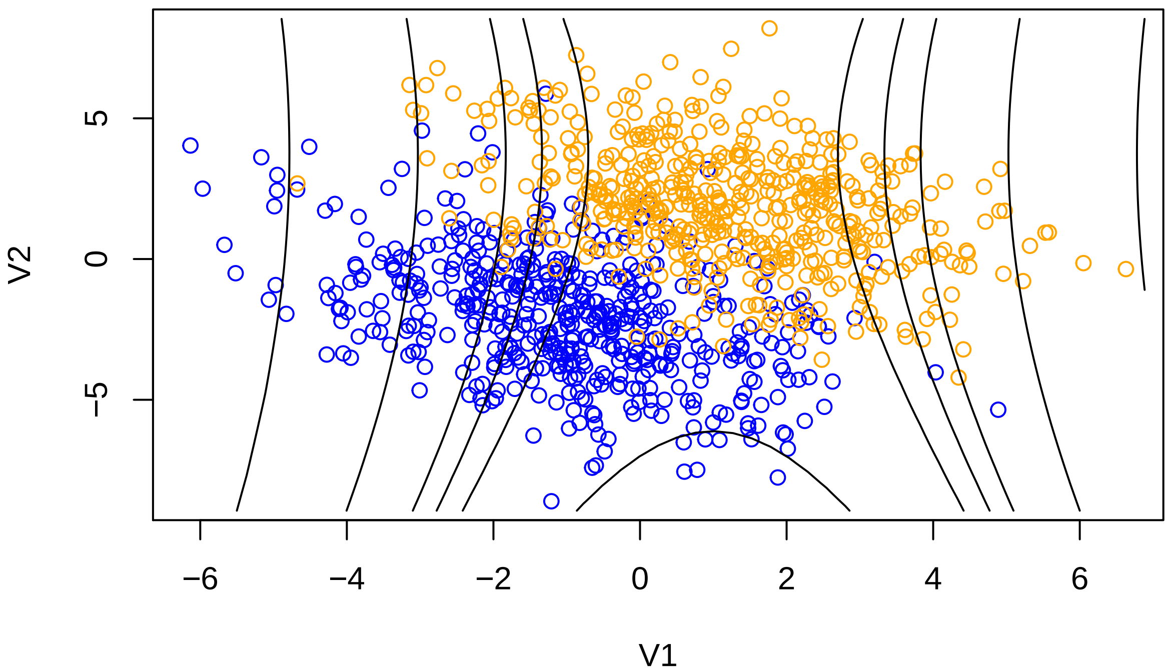

V2
V1
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4


V2
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4
V1


V2
V1
8
6
4
2
0
−2
−4
−6
−4
−2
0
2
4


5
0
−5
V2
−6
−4
−2
0
2
4
6
V1


V2
V1
5
0
−5
−6
−4
−2
0
2
4
6



2.2 Test sur données réelles

2.2.1 Données Pima

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

models = c(adq.app, adl.app, nba.app)
count = 0
models.names = c("ADP","ADL","NBA")
for (model in models) {
  count = count + 1
  print(models.names[count])
  model.res = ad.models(X,z,model,niter = 100)
  print(model.res$estErrApp)
  print(model.res$estErrTest)
  print("")
}
```

```
## [1] "ADP"
## [1] 0.3482817
## [1] 0.3466667
## [1] ""
## [1] "ADL"
```

```
## [1] 0.3345634
## [1] 0.3346328
## [1] ""
## [1] "NBA"
## [1] 0.3262817
## [1] 0.3286441
## [1] ""
```

2.2.2 Données `breast cancer Wisconsin`

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

models = c(adq.app, adl.app, nba.app)
count = 0
models.names = c("ADP","ADL","NBA")
for (model in models) {
  count = count + 1
  print(models.names[count])
  model.res = ad.models(X,z,model,niter = 100)
  print(model.res$estErrApp)
  print(model.res$estErrTest)
  print("")
}
```

```
## [1] "ADP"
## [1] 0.1090989
## [1] 0.1112281
## [1] ""
## [1] "ADL"
## [1] 0.2343956
## [1] 0.2339912
## [1] ""
## [1] "NBA"
## [1] 0.05648352
## [1] 0.05394737
## [1] ""
```

3 Règle de Bayes

Les données synthétiques traitées précédemment, on été obtenues avec un modèle génératif.

On va ici s'intéresser à la détermination analytique des frontière de décision.

3.1 Distributions marginales des variables X_1, X_2

Puis que $\mathbf{X}$ suit une loi normales multidimensionnelle conditonellement à Z :

$$\mathbf{X}|Z = k \sim \mathcal{N}(\mu_k, \Sigma_k)$$

On a marginalement des lois normales pour les composantes :

$$\forall i \in \{1,2\}, \quad X_i|Z=k \sim \mathcal{N}((\mu_k)_i, (\Sigma_k)_{ii})$$

3.2 Iso-densité via la log-densité:

Travailler avec la densité conditionnelle revient à travailler avec la log-densité conditionelle :

$$\log f_k(x) = -\frac{p}{2}\log(2\pi) - \frac{1}{2}\log\det\Sigma_k - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k)$$

On s'intéresse ici à la courbe d'iso-densité Γ_c avec $c \in [0,1]$; ie:

$$\begin{array}{rcc} x \in \Gamma_c & \Leftrightarrow & \log f_k(x) + \log \pi_k = \log c \\ & \Leftrightarrow & (x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) = 2\log c - p\log(2\pi) - \log\det\Sigma_k + 2\log\pi_k \\ & \Leftrightarrow & \|(x-\mu_k)\|^2_{\Sigma_k^{-1}} = C \end{array}$$

avec $C = -2\log c - p\log(2\pi) - \log\det\Sigma_k + +2\log\pi_k$

Ainsi

$$\Gamma_c \quad : \quad \|(x-\mu_k)\|^2_{\Sigma_k^{-1}} = C$$

avec $C = -2\log c - p\log(2\pi) - \log\det\Sigma_k + 2\log\pi_k$

Γ_c est :

— dans le cas général, une ellispsoïde de centre μ_k et de certains demis-axes (obtenables par diagonalisation)
— dans le cas diagonal, une ellispoïde de centre μ_k et de demis-axes σ_{k1}^{-2}, ..., σ_{kn}^{-2} centré sur les axes de la base canoniques
— dans le cas sphérique, une sphère de centre μ_k et de rayon σ_k^{-2}

3.3 Frontières de décision dans le cas général

$$\mathcal{F} = \{x \in \mathbb{R}^2 | \mathbb{P}(x|\omega_1) = \mathbb{P}(x|\omega_2)\}$$

On a successivement :

$$\begin{array}{rcc} x \in \mathcal{F} & \Leftrightarrow & -\frac{p}{2}\log 2\pi - \frac{1}{2}\log\det\Sigma_k - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log\pi_k = -\frac{p}{2}\log 2\pi - \frac{1}{2}\log\det\Sigma_l - \frac{1}{2}(x-\mu_l)^T\Sigma_l^{-1}(x-\mu_l) \\ & \Leftrightarrow & (x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) = (x-\mu_l)^T\Sigma_l^{-1}(x-\mu_l) + \log\frac{\det\Sigma_l}{\det\Sigma_k} + 2\log\frac{\pi_k}{\pi_l} \\ & \Leftrightarrow & x^T(\Sigma_k^{-1} - \Sigma_l^{-1})x + 2x^T(\Sigma_l^{-1}\mu_l - \Sigma_k^{-1}\mu_k) = \|\mu_l\|^2_{\Sigma_l^{-1}} - \|\mu_k\|^2_{\Sigma_k^{-1}} + \log\frac{\det\Sigma_l}{\det\Sigma_k} + 2\log\frac{\pi_k}{\pi_l} \end{array}$$

Pour réaliser le contour, on définit fonction suivante dans le cas général :

```
man = function(x, y, M) t(x-y) %*% M %*% (x-y)

f = function(x, y, mu1, mu2, S1, S2 ){
  S1_inv = solve(S1)
  S2_inv = solve(S2)
  n = length(x)
  res = rep(0, n)
```

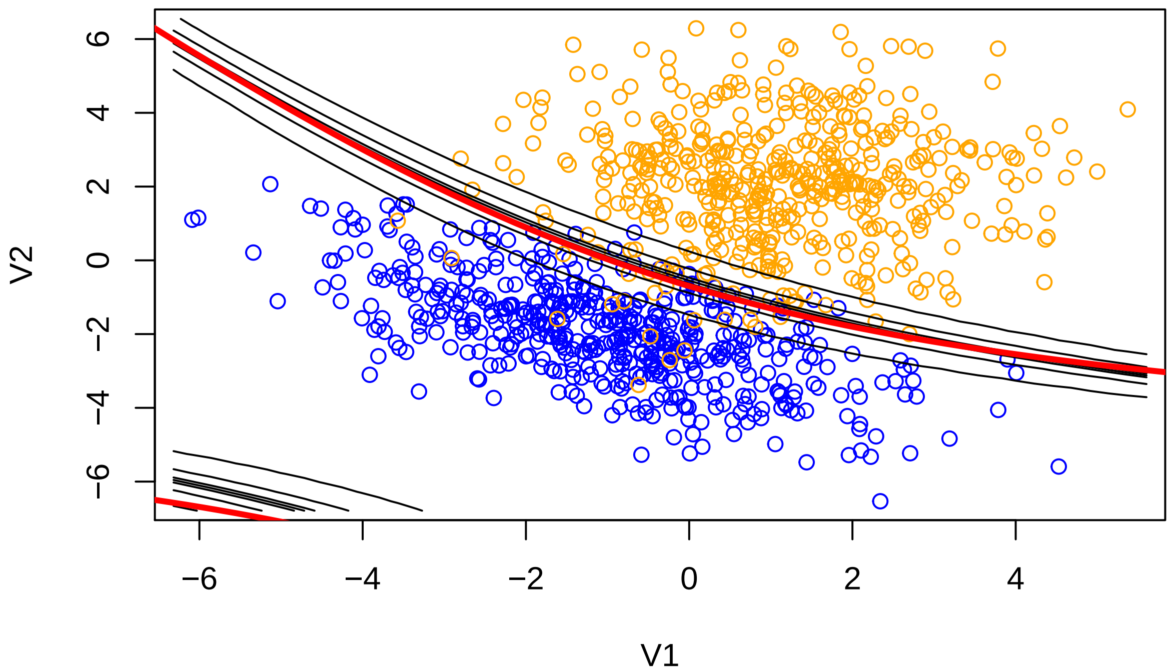
```
  for (i in 1:n){
    X = matrix(c(x[i], y[i]))
    res[i] = man(X, mu1, S1_inv)**2 - man(X, mu2, S2_inv)**2
  }
  res
}

# Niveaux pour isoproba
levels = c(0.1, 0.3, 0.45,0.5, 0.55,0.7, 0.9)
```

Avec ce que l'on a montré, on peut définit cette autre fonction pour le tracé :

```
g = function(x, y, mu1, mu2, S1, S2 ){
  S1_inv = solve(S1)
  S2_inv = solve(S2)
  n = length(x)
  res = rep(0, n)
  for (i in 1:n){
    X = matrix(c(x[i], y[i]))
    res[i] = man(X, 0*X, S1_inv - S2_inv)
    res[i] = res[i] + 2*t(X) %*% (S2_inv %*% mu2 - S1_inv %*% mu1)
    res[i] = res[i] - man(mu2,0*mu2,S2_inv) + man(mu1,0*mu1,S1_inv)
  }
  res
}
```

Celle-ci, couplée à `outer` puis `contour` permet de tracer la frontière de décision ici représentée en rouge sur les courbes d'isovaleurs déduite numériquement.

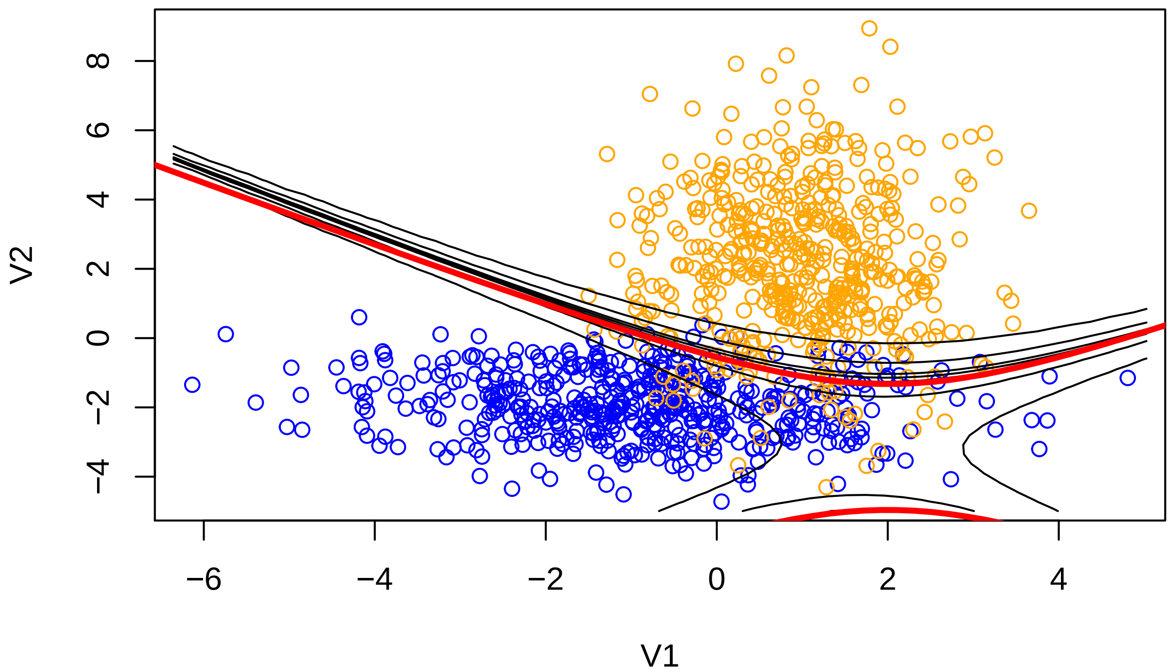
```
# Jeu de données Synth1-1000
dataSet = dataSets[1]
data = loadData(dataSet)
X = data$X
z = data$z

# Courbes d'iso proba
adq.params = adq.app(X, z)
prob.ad(adq.params, X,z, levels)


# Affichage de la frontière de décision
mu1 = matrix(c(-1, -2), nrow=2, ncol=1)
mu2 = matrix(c(1, 2), nrow=2, ncol=1)
S1 = matrix(c(3, -1.5, -1.5, 2), nrow=2, ncol=2, byrow=TRUE)
S2 = matrix(c(2, 0, 0, 3), nrow=2, ncol=2, byrow=TRUE)
x = y = seq(-10,10,length=100)
Z = outer(x,y,g,mu1,mu2,S1,S2)

contour(x=x, y=x, z=Z, levels=0, las=1, col="red", drawlabels=FALSE, lwd=3, add=TRUE)
```



3.4 Frontières de décision dans le cas diagonal

L'expression ici ne change pas. En revanche, on se retrouve avec une quadratique alignée selon les axes de la base canonique.

```
# Jeu de données Synth2-1000
dataSet = dataSets[2]
data = loadData(dataSet)
X = data$X
z = data$z

# Courbes d'iso proba
adq.params = adq.app(X, z)
prob.ad(adq.params, X,z, levels)

# Affichage de la frontière de décision
mu1 = matrix(c(-1, -2), nrow=2, ncol=1)
mu2 = matrix(c(1, 2), nrow=2, ncol=1)
S1 = matrix(c(3, 0, 0, 1), nrow=2, ncol=2, byrow=TRUE)
S2 = matrix(c(1, 0, 0, 4.5), nrow=2, ncol=2, byrow=TRUE)
x = y = seq(-10,10,length=100)
Z = outer(x,y,g,mu1,mu2,S1,S2)

contour(x=x, y=x, z=Z, levels=0, las=1, col="red", drawlabels=FALSE, lwd=3, add=TRUE)
```



3.5 Frontières de décision dans le cas d'égalité

On s'intéresse ici au cas $\Sigma_k = \Sigma_k = \Sigma$. Montrons que la frontière de décision est linéaire.

$$x^T 0x + 2x^T(\Sigma^{-1}(\mu_l - \mu_k)) = \|\mu_l\|^2_{\Sigma^{-1}} - \|\mu_k\|^2_{\Sigma^{-1}} + \log(1) + 2\log\frac{\pi_k}{\pi_l}$$

Soit:

$$\begin{cases} x^T d = c \\ d = 2\Sigma^{-1}(\mu_l - \mu_k) \\ c = \|\mu_l\|^2_{\Sigma^{-1}} - \|\mu_k\|^2_{\Sigma^{-1}} + 2\log\frac{\pi_k}{\pi_l} \end{cases}$$

La frontière de décision est donc ici un hyperplan.

```
h = function(x, y, mu1, mu2, S,void){
  S_inv = solve(S)
  n = length(x)
  res = rep(0, n)
  for (i in 1:n){
    X = matrix(c(x[i], y[i]))
    d = 2 * S_inv %*% (mu1 - mu2)
    c = man(mu1,0*mu1,S_inv) - man(mu2,0*mu2,S_inv)
    res[i] = t(X) %*% d - c
  }
```

```
  res
}
```

```
# Jeu de données Synth3-1000
dataSet = dataSets[3]
data = loadData(dataSet)
X = data$X
z = data$z

# Courbes d'iso proba
adq.params = adq.app(X, z)
prob.ad(adq.params, X,z, levels)

# Affichage de la frontière de décision
mu1 = matrix(c(-1, -2), nrow=2, ncol=1)
mu2 = matrix(c(1, 2), nrow=2, ncol=1)
S1 = matrix(c(3, -1.5, -1.5, 4.5), nrow=2, ncol=2, byrow=TRUE)
S2 = S1
x = y = seq(-10,10,length=100)
Z = outer(x,y,h,mu1,mu2,S1,S2)

contour(x=x, y=x, z=Z, levels=0, las=1, col="red", drawlabels=FALSE, lwd=3, add=TRUE)
```