TP4 - Discrimination

Chang QI (chang.qi@etu.utc.fr)

Valentin MONTUPET (valentin.montupet@etu.utc.fr)

UV : **SY09**

Semestre : **P17**

1. Programmation

1.1 Analyse discriminante

Dans cette partie, on se propose de programmer trois modèles d'analyse discriminante dans le cas binaire (jeux de données comptant g = 2 classes) : l'analyse discriminante quadratique, l'analyse discriminante linéaire, et le classifieur bayésien naïf. On complétera pour cela quatre fonctions : **adq.app**, **adl.app**, **nba.app** et **ad.val**.

Les trois premières fonctions réalisent l'estimation des paramètres $\widehat{\mu_k}, \widehat{\pi_k}, \widehat{\Sigma_k}$ respectivement de l'analyse discriminante quadratique (ADQ), l'analyse discriminante linéaire (ADL) et le classifieur bayésien naïf (NBA). En effet, en pratique, les paramètres μ_k, π_k, Σ_k du modèle sont inconnus mais grâce à l'ensemble d'apprentissage,on peut en déduire les estimateurs du maximum de vraisemblance (EMV). On rappelle les définitions de ces 3 paramètres pour chaque méthode de discrimination :

1.1.1 Analyse discriminante quadratique

$$\widehat{\pi_k} = \frac{n_k}{n}$$

$$\widehat{\mu_k} = \frac{1}{n_k} \sum_{i=1}^{n} z_{ik} x_i$$

$$\widehat{\Sigma_k} = V_k^* = \frac{1}{n_k - 1} \sum_{i=1}^{n} z_{ik} (x_i - \widehat{\mu_k})(x_i - \widehat{\mu_k})^T$$

où la variable z_{ik} vaut 1 si $x_i \in \omega_k$, 0 sinon. Pour l'estimateur $\widehat{\Sigma_k}$, on prend V_k^* qui est un estimateur sans biais. La fonction **adq.app** est donnée en annexe.

1.1.2 Analyse discriminante linéaire

On suppose cette fois que la matrice de covariance est commune à toutes les classes (hypothèse d'homoscédasticité).

$$\widehat{\pi_k} = \frac{n_k}{n}$$

$$\widehat{\mu_k} = \frac{1}{n_k} \sum z_{ik} x_i$$

$$\widehat{\Sigma} = V_W^* = \frac{1}{n - g} \sum_{k=1}^{g} (n_k - 1) V_k^*$$

La fonction **adl.app** est donnée en annexe.

1.1.3 Classifieur bayésien naïf

Ici, on suppose l'indépendance des variable X^j conditionnellement à Z ce qui revient dans un modéle gaussien à supposer les matrices Σ_k diagonales. On obtient donc une variante de l'ADQ où l'on estime les matrices de covariances par la matrice diagonale.

$$\widehat{\pi_k} = \frac{n_k}{n}$$

$$\widehat{\mu_k} = \frac{1}{n_k} \sum z_{ik} x_i$$

$$\widehat{\Sigma_k} = diag(s_{k1}^2, ..., s_{kj}^2, ..., s_{kp}^2)$$

La fonction **nba.app** est donnée en annexe.

1.1.4 Probabilités a posteriori

La fonction **ad.val** permet de calculer les probabilités a posteriori pour un ensemble de donner **Xtst** et d'effectuer le classement en fonction de ces dernières. Les probabilités a posteriori sont calculées comme suit :

$$P(Z = \omega_k | X = x) = \frac{f_k(x)\pi_k}{\pi_1 f_1(x) + \pi_2 f_2(x)}$$

La fonction **ad.val** est donnée en annexe.

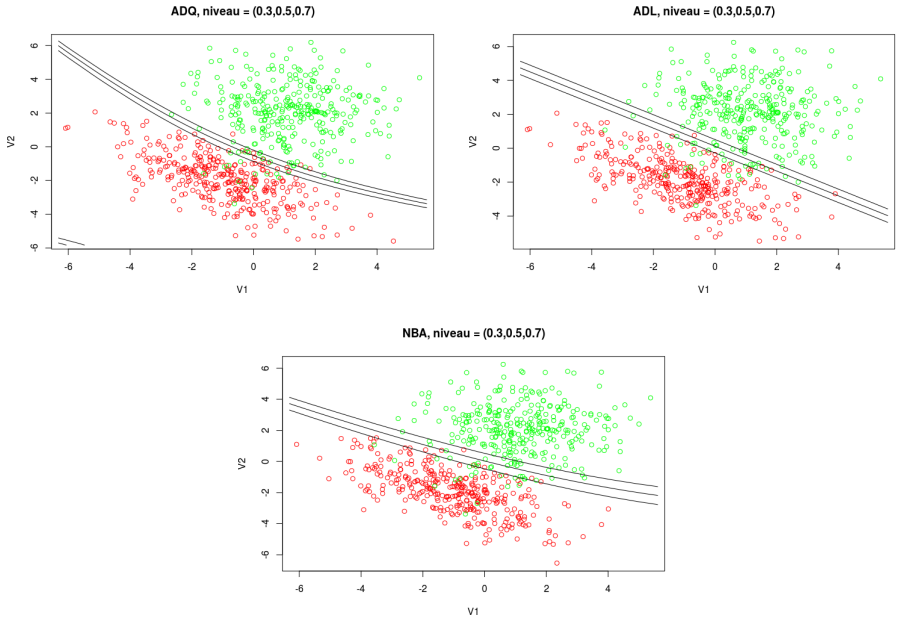

FIGURE 1.1 – Frontières de décision pour les données Synth1

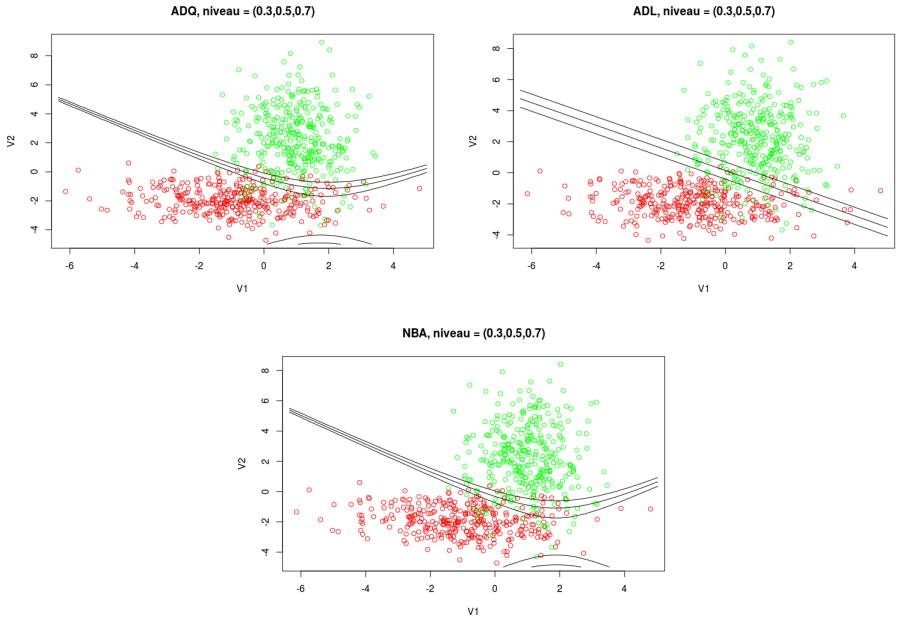

FIGURE 1.2 – Frontières de décision pour les données Synth2

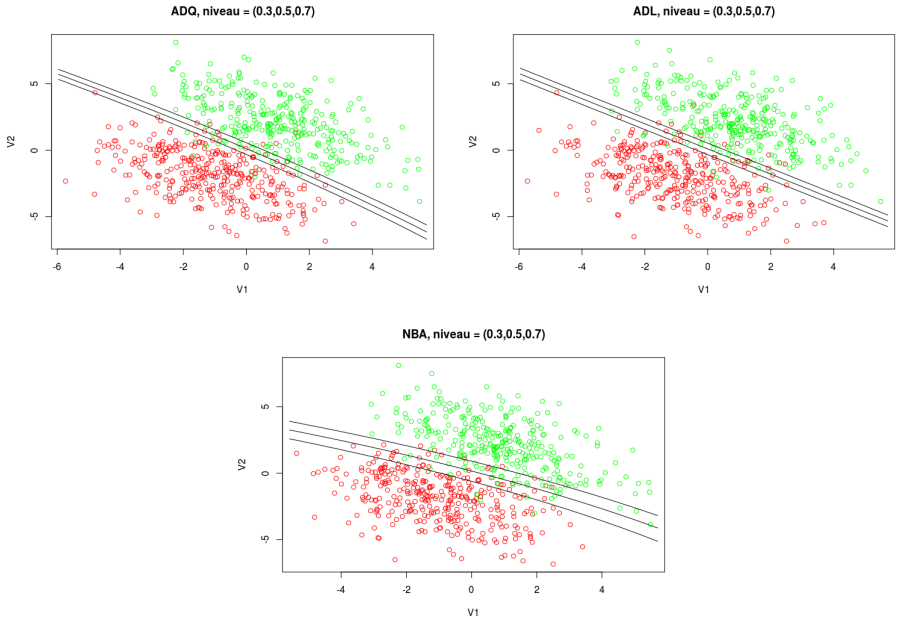

FIGURE 1.3 – Frontières de décision pour les données Synth3

1.2 Régression logistique

On se propose dans cette partie d'implémenter deux types de régression logistique : linéaire et quadratique. Dans les deux cas, nous utiliserons l'algorithme de Newton-Raphson afin d'apprendre les paramètres du modèle puis nous appliquerons le modèle obtenu sur un ensemble de données de test.

1.2.1 Apprentissage du modèle : log.app

Cette fonction permet d'apprendre les paramètres du modèle. Elle prend en entrée l'ensemble d'application **Xapp** et ses étiquettes **zapp** ainsi qu'un booléen indiquant s'il faut ou non ajouter une ordonnée à l'origine à la matrice d'exemple et un seuil **epsi** pour déterminer que la convergence de l'algorithme a bien eu lieu.

Elle retourne le vecteur β vecteur des poids calculé grâce à l'algorithme de Newton-Raphson : On initialise $\beta^{(0)}$ au vecteur nul de longueur p*1 ou (p+1)*1 si l'on ajoute une ordonnée à l'origine, puis pour chaque itération $q > 0$:

- On calcule les probabilités a posteriori $p^{(q)} = \frac{\exp(\beta^{(q-1)T} Xapp)}{1+\exp(\beta^{(q-1)T} Xapp)}$
- On calcule $W^{(q)}$ la matrice diagonale de terme générale $W_{ii}^{(q)} = p_i^{(q)}(1 - p_i^{(q)})$
- On met à jour β avec $\beta^{(q)} = \beta^{(q-1)} + (X_{app}^T W^{(q)} X_{app})^{-1} X_{app}^T (z_{app} - p^{(q)})$
- On regarde si $|\beta^{(q-1)} - \beta^{(q)}| < \epsilon$, si c'est le cas, on arrête l'algorithme et on renvoie les paramètres suivants :

le vecteur des poids final β, le nombre d'itérations exécutées par l'algorithme et la valeur LogL de la vraisemblance à l'optimum.

La fonction **log.app** est donnée en annexe.

1.2.2 Classement : log.val

La fonction prend en entrée les individus à classer **Xtst** et le vecteur des poids β obtenu avec la fonction précédente. La fonction se charge de calculer les probabilités à posteriori $P(w_1|x) = \frac{\exp(\beta^T Xtst)}{1+\exp(\beta^T Xtst)}$ et $P(w_2|x) = 1 - P(w_1|x)$ Enfin, pour chaque individu, nous pouvons prédire son étiquette en choisissant la probabilité a posteriori la plus élevée. La fonction renvoie finalement le vecteur des probabilités a posteriori *prob* et les étiquettes prédites *pred*.

La fonction **log.val** est donnée en annexe.

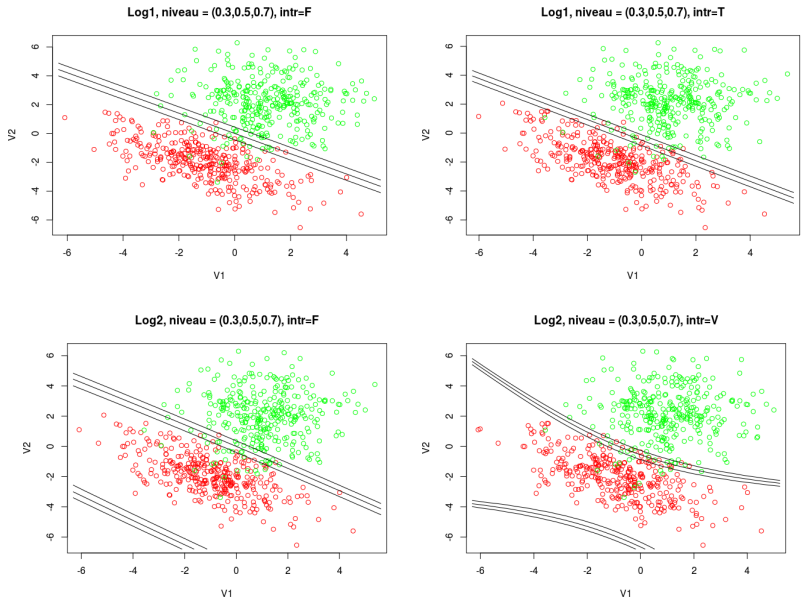

FIGURE 1.4 – Frontières de décision pour les données Synth1 avec la régression logistique linéaire (en haut) et quadratique (en bas)

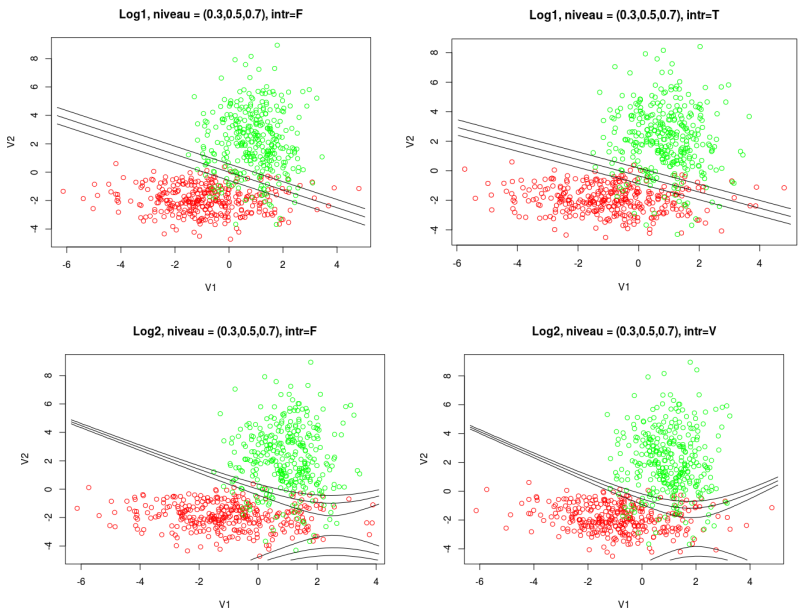

FIGURE 1.5 – Frontières de décision pour les données Synth2 avec la régression logistique linéaire (en haut) et quadratique (en bas)

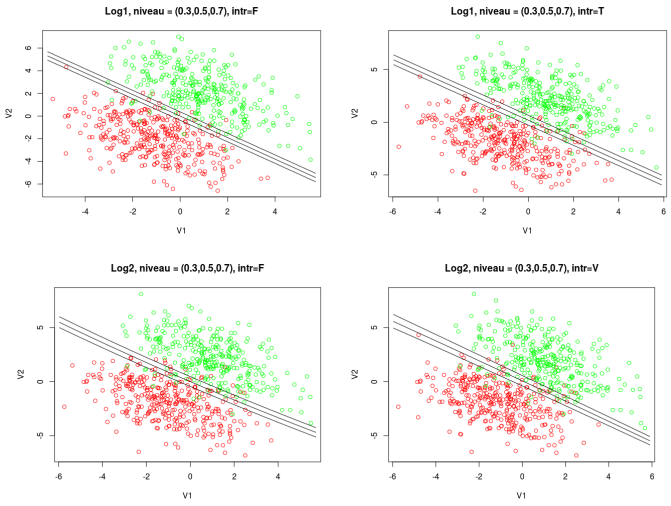

FIGURE 1.6 – Frontières de décision pour les données Synth3 avec la régression logistique linéaire (en haut) et quadratique (en bas)

1.2.3 Régression logistique quadratique

La régression logistique quadratique est une généralisation de la régression linéaire puisque le procédé est identique, la seule différence résidant dans les données elles-mêmes.

En effet, il s'agit de transformer les données $(X^1, X^2, .., X^P)$ en un espace de dimension plus grande, par exemple : $\chi^2 = (X^1, X^2, .., X^P, X^1X^2, ..., X^{P-1}X^P, (X^1)^2, ..., (X^P)^2)$. En conséquence, le modèle obtenu est plus flexible mais peut s'avérer moins robustes du fait du grand nombre de paramètres à estimer. Nous avons donc implémenter une fonction permettant de calculer χ^2, puis nous avons appliquer la régression logistique comme précédemment sur cette nouvelle matrice.

2. Application

2.1 Test sur données simulées

Nous cherchons à comparer les performances de l'analyse discriminante, de la régression logistique et des arbres de décision sur les jeux de données **Synth1-1000**, **Synth2-1000** et **Synth3-1000**. Pour ce faire, nous répéterons 20 fois le protocole suivant :

- Séparation aléatoire des données en un ensemble d'apprentissage et un ensemble de test
- Apprentissage du modèle sur l'ensemble d'apprentissage
- Classement des données de test et calculer le taux d'erreur associé.

Il s'agit ensuite de faire la moyenne des 20 taux d'erreur obtenus :

	Synth1-1000	Synth2-1000	Synth3-1000
ADQ	3,24%	6,36%	4,00%
ADL	4,52%	7,93%	3,94%
NBA	4,23%	6,25%	5,10%
Tree	4,47%	7,38%	6,69%
Log Lin (intr=F)	4,14%	7,38%	4,24%
Log Lin (intr=T)	3,27%	7,10%	4,23%
Log Quad (intr=F)	4,40%	6,95%	4,12%
Log Quad (intr=T)	3,47%	6,58%	4,20%

TABLE 2.1 – Taux d'erreur de l'analyse discriminante, de la régression logistique et de l'arbre sur les différents jeux de données

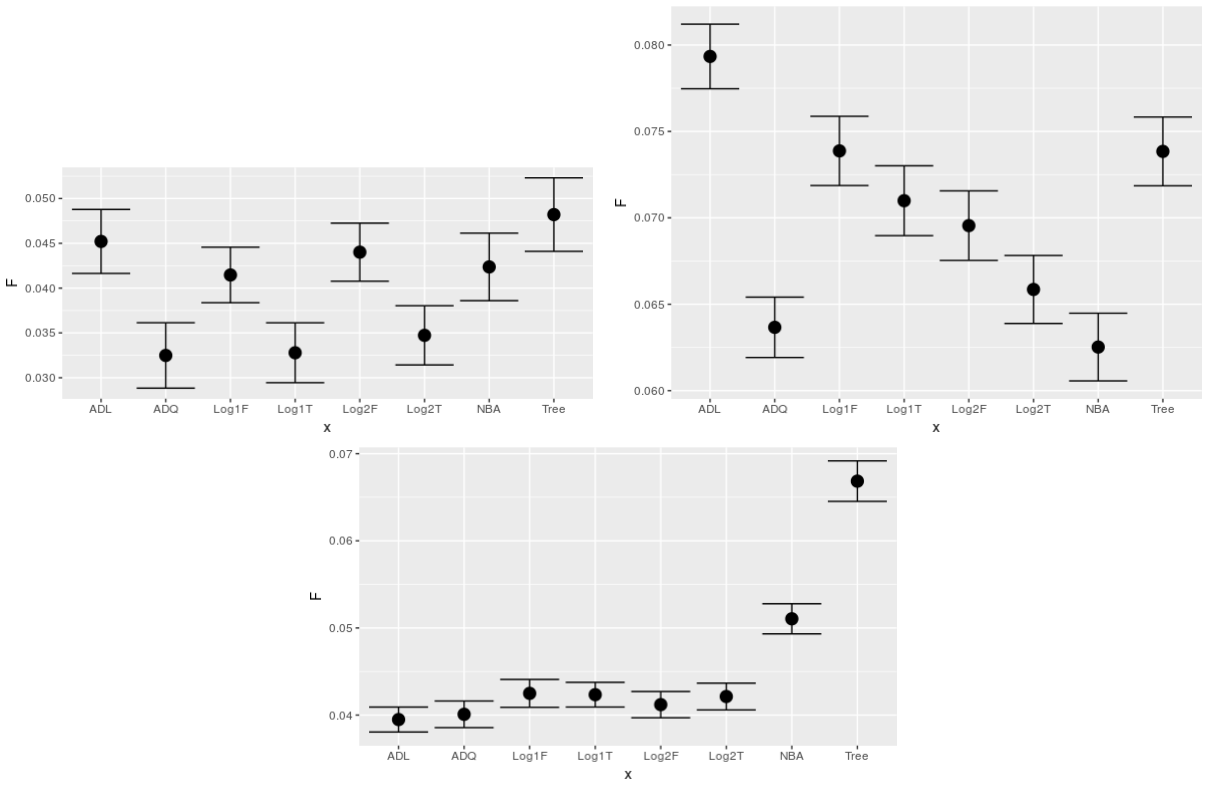

FIGURE 2.1 – Intervalle de confiance des taux d'erreurs pour les données Synth1 (haut-gauche), Synth2 (haut-droite), Synth3 (bas)

On peut constater que ces taux d'erreur sont cohérents avec les différentes frontières de décisions obtenues sur les figures précédentes. De manière générale, les résultats sont plutôt bons car les données suivent pour chaque classe une loi normale multivariée.

Concernant le jeu de données **Synth1**, nous constatons que l'analyse discriminante quadratique nous fournit le meilleur taux d'erreur de tous les classifieurs, suivi de la régression logistique avec ordonnée à l'origine. De plus, on sait que Σ_2 est diagonale, mais pas Σ_1. NBA supposant la diagonalité de ces matrices, on aurait pu penser qu'il fonctionnerait plutôt bien, mais en réalité il ne fonctionne que moyennement. De manière générale, ajouter une ordonnée à l'origine améliore considérablement le taux d'erreur de la régression logistique, traduisant le fait que les données ne sont sûrement pas centrées autour de l'origine.

Concernant le jeu de données **Synth2**, les matrices de covariances sont différentes avec $\Sigma_1 \simeq \left(\begin{smallmatrix} 3 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ et $\Sigma_2 \simeq \left(\begin{smallmatrix} 1 & 0 \\ 0 & 5 \end{smallmatrix}\right)$. Il n'est donc pas étonnant de voir l'ADL et la régression logistique linéaire proposer le taux d'erreur le plus élevé, puisque ces méthodes supposent l'identité des matrices de covariance. Au contraire, le classifieur Bayésien naïf qui suppose les matrices de covariances diagonales propose un bon résultat pour ce jeu de données.

Concernant le jeu de données **Synth3**, nous remarquons que les meilleurs résultats proviennent de l'ADL. Ce résultat n'est pas surprenant car l'on constate que les deux classes ont à peu près la même matrice de covariance Σ et sont répartis autour de l'origine (0,0) : le cas idéal pour ce classifieur. Exceptés NBA et Tree qui proposent des taux d'erreurs trop élevés, tous les autres se tiennent dans un mouchoir de poche.

2.2 Test sur données réelles

2.2.1 Données Pima

Classifieur	Taux erreur moyen
ADQ	24,49%
ADL	22,72%
NBA	23,08%
Log Lin (intr=F)	28,66 %
Log Lin (intr=T)	22,00%
Log Quad (intr=F)	25,21%
Log Quad (intr=T)	24,81%
Tree	25,48 %

TABLE 2.2 – Performance des classifieurs sur les données Pima : comparaison des taux d'erreur moyens pour N=100 exécutions

On constate que les taux d'erreur sont bien plus élevés sur ce jeu de données que sur les données simulées : on en déduit qu'aucun classifieur ne permet de construire une frontière de décision suffisante pour distinguer les deux classes. Pourtant, comme nous l'avons vu précédemment, les classifieurs produisaient de très bons résultats. On peut supposer que ce phénomène vient alors des données en elles mêmes. Cette supposition se confirme lorsque l'on trace la matrice de graphique en fonctions de toutes les dimensions dans R : aucune variable ne permet de distinguer clairement les deux classes.

2.2.2 Données Breastcancer Wisconsin

Classifieur	Taux erreur moyen
ADQ	5,05%
ADL	4,55%
NBA	3,76%
Log Lin (intr=F)	15,62 %
Log Lin (intr=T)	4,08%
Tree	5,30%

TABLE 2.3 – Performance des classifieurs sur les données Breastcancer Wisconsin : comparaison des taux d'erreur moyens pour N=100 exécutions

Nous obtenons ici de très bons résultats. On remarque de suite que la régression logistique linéaire sans ordonnées à l'origine propose un taux d'erreur médiocre par rapport à toutes les autres méthodes : on peut se douter que les classes ne sont pas centrées par rapport à l'origine (0,0). Ce doute se confirme aisément, puisque les valeurs des données sont toutes des entiers strictement supérieurs à 0. Le classifieur NBA, qui suppose les matrices de covariances diagonales, fournit le meilleur taux d'erreur. En constant les matrices de covariances de chaque classe, on constate que l'hypothèse est plutôt justifié puisque les

poids des termes diagonaux sont respectivement de 36,2% et 43,5% alors que ces derniers ne représentent que $9/81 = 11\%$ des valeurs de la matrice.

```
> round(cov(w1),1)
     V2  V3  V4  V5  V6  V7  V8  V9 V10
V2  2.8 0.4 0.5 0.4 0.2 0.2 0.2 0.3 0.0
V3  0.4 0.7 0.6 0.2 0.3 0.5 0.2 0.4 0.0
V4  0.5 0.6 0.9 0.2 0.3 0.4 0.2 0.4 0.0
V5  0.4 0.2 0.2 0.8 0.2 0.4 0.1 0.2 0.0
V6  0.2 0.3 0.3 0.2 0.8 0.3 0.1 0.4 0.0
V7  0.2 0.5 0.4 0.4 0.3 1.5 0.3 0.3 0.1
V8  0.2 0.2 0.2 0.1 0.1 0.3 1.1 0.3 0.0
V9  0.3 0.4 0.4 0.2 0.4 0.3 0.3 0.9 0.0
V10 0.0 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.3
> sum(diag(round(cov(w1),1)))/sum(round(cov(w1),1))
[1] 0.362963
```

```
> round(cov(w2),1)
      V2   V3   V4   V5   V6   V7   V8   V9 V10
V2   5.9  0.6  0.7 -1.1  0.1 -0.4 -0.1 -0.1 0.7
V3   0.6  7.4  5.0  2.8  3.1 -0.4  2.4  2.7 1.7
V4   0.7  5.0  6.6  2.2  2.4 -0.2  2.0  2.7 1.4
V5  -1.1  2.8  2.2 10.2  1.5 -1.7  2.5  2.0 1.6
V6   0.1  3.1  2.4  1.5  6.0 -0.2  1.2  1.9 2.1
V7  -0.4 -0.4 -0.2 -1.7 -0.2  7.1 -0.6  0.3 0.0
V8  -0.1  2.4  2.0  2.5  1.2 -0.6  5.2  1.9 0.3
V9  -0.1  2.7  2.7  2.0  1.9  0.3  1.9 11.2 1.9
V10  0.7  1.7  1.4  1.6  2.1  0.0  0.3  1.9 6.6
> sum(diag(round(cov(w2),1)))/sum(round(cov(w2),1))
[1] 0.4355263
```

FIGURE 2.2 – Matrices de covariances des deux classes arrondies à 1 décimale, et pourcentage poids des termes diagonaux

2.3 Challenge : données SPAM

Analysons tout d'abord la nature de ce jeu de données : il comporte 4601 individus : 1813 Spam (39.4%) et 2788 Non Spam (60.6%). Chaque individu est caractérisé par 57 variables aléatoires quantitatives : les 54 premières semblent être un pourcentage compris entre $[0, 100]$. Après recherche sur le web [1], nous apprenons que les 48 premières variables représentent la fréquence d'un mot donné dans l'email et les 6 autres **la fréquence d'un caractère donné (ex : !, ?, , $) dans un email.**. Les 3 autres variables, prenant des valeurs plus élevées, correspondent à la **longueur moyenne des séquences de lettres capitales ininterrompues, la longueur de la plus longue séquence de lettre capitales ininterrompues et le nombre de lettres capitales dans l'email**.

L'idée ici est de trouver un classifieur qui minime le taux de faux positifs, qui est l'action de classer un véritable email comme spam, tout en recherchant la meilleure classification possible des spams en tant que tels.

Pour évaluer la performance d'un classifieur sur ce jeu de données, nous pouvons définir :

- Positif (P) : Nombre total de spams
- Négatif (N) : Nombre total de non-spams
- Vrai Positif (VP) : Nombre de spams classés en tant que spams
- Vrai Négatif (VN) : Nombre de non-spams classés en tant que non-spams
- Faux Positif (FP) : Nombre de non-spams classés en tant que spams
- Faux Négatif (FN) : Nombre de spams classés en tant que non-spams
- Justesse :(VP+VN)/(P+N)
- Précision :VP/(VP+FP)
- Rappel : VP/(VP+FN)

On peut les représenter dans une matrice de confusion, mais il est plus intéressant de tracer la courbe de ROC pour comparer la performance de différents classifieurs.

1. https ://archive.ics.uci.edu/ml/datasets/Spambase

2.3.1 Courbe ROC

La courbe ROC est un outil d'évaluation et de comparaison des modèles. Ici, on prend la spécificité (1-FP) comme axe X et la sensibilité (1-FN) comme axe Y. On peut aussi calculer l'aire compris entre l'axe x et la courbe ROC, appellée AUC (Area Under Curve) : plus la surface est grande, plus le classifieur est performant, puisque la courbe ROC se rapproche davantage du point (1,1) le cas idéal. Ici, nous avons appliqué les méthodes de LDA et QDA au jeu de données SPAM, et nous avons réalisé la validation croisée par 10-folds. Les courbes ROC obtenus sont obtenus figure 2.3.

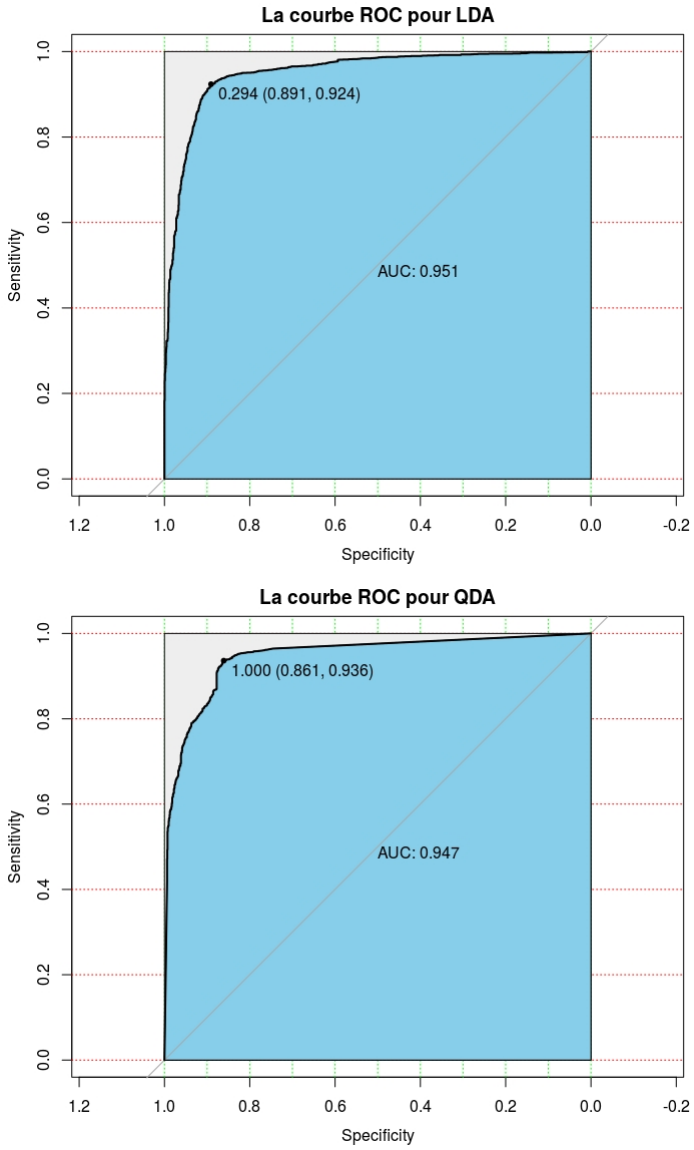

FIGURE 2.3 – La courbe ROC pour différents classifieurs

On obtient de très bons résultats pour les deux classifieurs avec un très léger avantage à l'analyse discriminante linéaire puisque l'AUC est un petit peu plus élevé. Nous avons essayé d'implémenter la régression logistique sans succès : du fait du grand nombre de variables disponibles, la matrice $(X_{app}^T W^{(q)} X_{app})$ n'est pas inversible.

2.3.2 Arbres

Nous nous proposons enfin d'appliquer les arbres de décision au jeu de données. Nous obtenons un **taux d'erreur de 10,2%** grâce au classifieur préalablement utilisé. Ce taux d'erreur reste plutôt insuffisant, c'est pourquoi nous essayerons la méthode « Random Forest » (via le package éponyme). en lançant la fonction $randomForest(x = training[, 1 : 57], y = as.factor(training\$z))$, avec un nombre de 500 arbres, nous obtenons **un taux d'erreur moyen de 5,07%** pour les données de test.

		Prédiction	Prédiction	Taux erreur de classe
		Positif	Négatif	
Réalité	Positif	1155	96	7,67%
Réalité	Négatif	63	1907	3,19%

TABLE 2.4 – Matrice de confusion pour l'ensemble d'apprentissage

		Prédiction	Prédiction	Taux erreur de classe
		Positif	Négatif	
Réalité	Positif	514	48	9,33%
Réalité	Négatif	25	793	3,15%

TABLE 2.5 – Matrice de confusion pour l'ensemble de test

On remarque donc que la méthode Random Forest est très efficace : nous avons très peu de faux négatifs et de faux positifs. La fonction est donnée en annexe.

Conclusion

Ce TP nous a donc permis d'appliquer les différentes méthodes d'analyse discriminante et de régression logistique sur des ensembles de données simulée d'une part et réelles d'autre part. Nous avons constaté l'efficacité des méthodes de discrimination lorsque les données simulées répondent à certains critères, mais nous avons constaté que la réalité est plus compliquée et que tout n'est pas comme on le souhaiterait : données qui ne suivent pas forcément une distribution normale, distribution inconnue, etc.

A. adq.app

```
adq.app <- function(Xapp, zapp){
    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)

        param$MCov[,,k] <- cov(Xapp[indk[],])*(length(indk)/(length(indk)-1))
        param$mean[k,] <- apply(Xapp[indk[],],2,sum)/length(indk)
        param$prop[k] <- length(indk)/n
    }
    param
}
```

B. adl.app

```
adl.app <- function(Xapp, zapp){
    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)

        MCov <- MCov + cov(Xapp[indk[],])*length(indk)
        param$mean[k,] <- apply(Xapp[indk[],],2,sum)/length(indk)
        param$prop[k] <- length(indk)/n
    }
    MCov <- MCov/(n-g)
    for (k in 1:g){
        param$MCov[,,k] <- MCov
    }
    param
}
```

C. nba.app

```
nba.app <- function(Xapp, zapp){
    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)

        param$MCov[,,k] <- diag(diag(cov(Xapp[indk[],])))
        param$mean[k,] <- apply(Xapp[indk[],],2,sum)/length(indk)
        param$prop[k] <- length(indk)/n
    }
    param
}
```

D. ad.val

```
ad.val <- function(param, Xtst){
  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){
    prob[,k] <- param$prop[k]*mvdnorm(Xtst, param$mean[k,], param$MCov[,,k])
  }
  prob <- prob/(param$prop[1]*mvdnorm(Xtst, param$mean[1,], param$MCov[,,1])
    +param$prop[2]*mvdnorm(Xtst, param$mean[2,], param$MCov[,,2]))
  pred <- max.col(prob)

  out$prob <- prob
  out$pred <- pred
  out
}
```

E. log.app & log.val

```
log.app <- function(Xapp, zapp, intr, epsi){
    n <- dim(Xapp)[1]
    p <- dim(Xapp)[2]
    Xapp <- as.matrix(Xapp)

    if (intr == T){
        Xapp <- cbind(rep(1,n),Xapp)
        p <- p + 1
    }
    targ <- matrix(as.numeric(zapp),nrow=n)
    targ[which(targ==2),] <- 0
    tXap <- t(Xapp)
    beta <- matrix(0,nrow=p,ncol=1)
    conv <- F
    iter <- 0
    while (conv == F){
        iter <- iter + 1
        bold <- beta
        prob <- postprob(beta, Xapp)
        MatW <- diag(c(prob*(1-prob)))
        beta <- beta + solve(tXap%*%MatW%*%Xapp)%*%tXap%*%(targ-prob)

        if (norm(beta-bold)<epsi){
            conv <- T
        }
    }
    prob <- postprob(beta, Xapp)
    out <- NULL
    out$beta <- beta
    out$iter <- iter
    out$logL <- tXap%*%(targ-prob)
    out
}
```

```
log.val <- function(beta, Xtst){
    m <- dim(Xtst)[1]
    p <- dim(beta)[1]
    pX <- dim(Xtst)[2]
    Xtst <- as.matrix(Xtst)

    if (pX == (p-1)){
        Xtst  <- cbind(rep(1,m),Xtst)
    }
    prob <- postprob(beta,Xtst)
    pred <- max.col(cbind(prob,1-prob))
    out <- NULL
    out$prob <- prob
    out$pred <- pred
    return(out)
}

postprob <- function(beta, X){
    X <- as.matrix(X)
    tmp <- exp(X%*%beta)
    prob <-tmp/(1+tmp)
    prob
}
```

F. Random Forest

```
TauxErreurRF <- function(data,N){
  error<-matrix(0,N,1)
  p <- ncol(data)
  for (i in 1:N){
    dat<-separ1(data[,-p],data[,p])
    Xapp <- dat$Xapp
    zapp <- dat$zapp
    Xtst <- dat$Xtst
    ztst <- dat$ztst
    nb_tst<-length(dat$ztst)
    rf <- randomForest(x = Xapp, y = as.factor(zapp))
    zpred <- predict(rf, Xtst)
    for(j in 1:nb_tst)
    {
      if (zpred[j] != ztst[j])
      {
        error[i,1] <- error[i,1] + 1
      }
    }
    error[i,1] <- error[i,1]/nb_tst
  }
  error
}
```