Apprentissage supervisé avec R

Laurent Rouvière

6 janvier 2025

Table des matières

Présentation

Ce tutoriel présente les principaux algorithmes d'**apprentissage supervisé avec R**. On pourra trouver :

- les supports de cours associés à ce tutoriel ainsi que les données utilisées dans le dépôt gitlab https://plmlab.math.cnrs.fr/rouviere/TUTO_ML ;
- le tutoriel sans les correction à l'url https://rouviere.pages.math.cnrs.fr/TUTO_ML/
- le tutoriel avec les corrigés (à certains moment) à l'url https://rouviere.pages.math.cnrs.fr/TUTO_ML/correction/.

Les thèmes suivants sont abordés :

- **Estimation du risque** : présentation du metapackage `tidymodels` ;
- **Méthodes linéaires** : analyse discriminante et SVM ;
- **Arbres et agrégation d'arbres** : méthode CART, forêts aléatoires et gradient boosting ;
- **Réseaux de neurones et introduction au deep learning**, perceptron multicouches avec `keras` ;
- **Compléments** : données déséquilibrées et comparaison d'algorithmes.

Il existe de nombreuses références sur le machine learning, la plus connue étant certainement Hastie, Tibshirani, et Friedman (2009), disponible en ligne à l'url https://web.stanford.edu/~hastie/ElemStatLearn/. On pourra également consulter Boehmke et Greenwell (2019) qui propose une présentation très claire des algorithmes machine learning avec **R**. Cet ouvrage est également disponible en ligne à l'url https://bradleyboehmke.github.io/HOML/.

partie I

Notion de risque en apprentissage

1 Calcul de risques avec R

L'apprentissage supervisé consiste à expliquer ou prédire une sortie $y \in \mathcal{Y}$ par des entrées $x \in \mathcal{X}$ (le plus souvent $\mathcal{X} = \mathbb{R}^p$). Cela revient à trouver un **algorithme** ou **machine** représenté par une fonction

$$f : \mathcal{X} \to \mathcal{Y}$$

qui à une nouvelle observation x associe la prévision $f(x)$. Bien entendu le problème consiste à chercher le **meilleur algorithme** pour le cas d'intérêt. Cette notion nécessite de meilleur algorithme la définition de **critères** que l'on va chercher à optimiser. Les critères sont le plus souvent définis à partir du fonction de perte

$$\begin{aligned} \ell : \mathcal{Y} \times \mathcal{Y} &\mapsto \mathbb{R}^+ \\ (y, y') &\to \ell(y, y') \end{aligned}$$

où $\ell(y, y')$ représentera l'erreur (ou la perte) pour la prévision y' par rapport à l'observation y. Si on représente le phénomène d'intérêt par un couple aléatoire (X, Y) à valeurs dans $\mathcal{X} \times \mathcal{Y}$, on mesurera la performance d'un algorithme f par son risque

$$\mathcal{R}(f) = \mathbf{E}[\ell(Y, f(X))].$$

Trouver le meilleur algorithme revient alors à trouver f qui minimise $\mathcal{R}(f)$. Bien entendu, ce cadre possède une utilité limitée en pratique puisqu'on ne connaît jamais la loi de (X, Y), on ne pourra donc jamais calculer le **vrai risque** d'un algorithme f. Tout le problème va donc être de trouver l'algorithme qui a le plus petit risque à partir de n observations $(x_1, y_1), \ldots, (x_n, y_n)$.

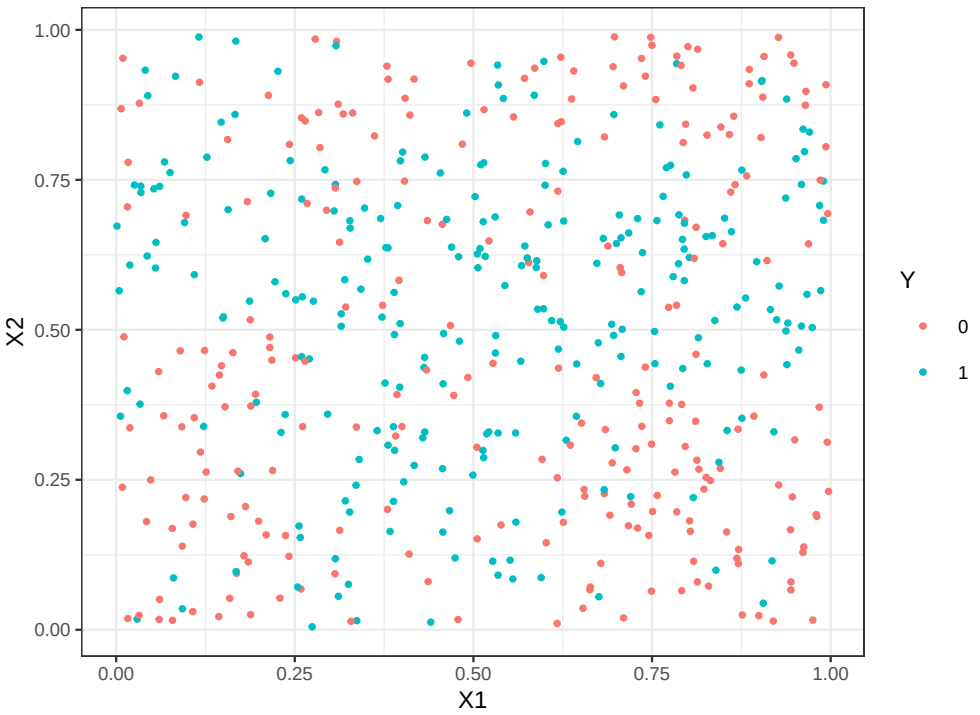
Nous verrons dans les chapitres suivants plusieurs façons de construire des algorithmes mais, dans tous les cas, un algorithme est représenté par une fonction

$$f_n : \mathcal{X} \times (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{Y}$$

qui, pour une nouvelle donnée x, renverra la prévision $f_n(x)$ calculée à partir de l'échantillon qui vit dans $(\mathcal{X} \times \mathcal{Y})^n$. Dès lors la question qui se pose est de calculer (ou plutôt d'estimer) le risque (inconnu) $\mathcal{R}(f_n)$ d'un algorithme f_n. Les techniques classiques reposent sur des algorithmes de type validation croisée. Nous les mettons en œuvre dans cette partie pour un algorithme simple : les k plus proches voisins. On commencera par programmer ces techniques "à la main" puis on utilisera le package **tidymodels** qui permet de calculer des risques pour quasiment tous les algorithmes que l'on retrouver en apprentissage supervisé.

Dans cette partie nous considérerons les données suivantes :

```r
gen_class_bin2D <- function(n=100,graine=1234,bayes=0.1){
  set.seed(graine)
  grille <- 0.1
  X1 <- runif(n)
  X2 <- runif(n)
  Y <- rep(0,n)
  cond0 <- (X1>0.2 & X2>=0.8) | (X1>0.6 & X2<0.4) | (X1<0.25 & X2<0.5)
  cond1 <- !cond0
  Y[cond0] <- rbinom(sum(cond0),1,bayes)
  Y[cond1] <- rbinom(sum(cond1),1,1-bayes)
  donnees <- tibble(X1,X2,Y=as.factor(Y))
  px1 <- seq(0,1,by=grille)
  px2 <- seq(0,1,by=grille)
  px <- expand.grid(X1=px1,X2=px2)
  py <- rep(0,nrow(px))
  cond0 <- (px[,1]>0.2 & px[,2]>=0.8) | (px[,1]>0.6 & px[,2]<0.4) | (px[,1]<0.25 & px[,2]<0.5)
  cond1 <- !cond0
  py[cond0] <- 0
  py[cond1] <- 1
  df <- px |> as_tibble() |> mutate(Y=as.factor(py))
  p <- ggplot(df)+aes(x=X1,y=X2,fill=Y)+geom_raster(hjust=1,vjust=1)#+theme(legend.position='none')
  return(list(donnees=donnees,graphe=p))
}
donnees <- gen_class_bin2D(n=500,graine=12345,bayes=0.20)$donnees
head(donnees)
## # A tibble: 6 x 3
##      X1    X2 Y
##   <dbl> <dbl> <fct>
## 1 0.721 0.209 0
## 2 0.876 0.766 1
## 3 0.761 0.842 1
## 4 0.886 0.934 0
## 5 0.456 0.676 0
## 6 0.166 0.859 1
ggplot(donnees)+aes(x=X1,y=X2,color=Y)+geom_point()
```



Le problème est d'expliquer/prédire une variable binaire Y par deux variables quantitatives X_1 et X_2.

1.1 L'algorithme des k plus proches voisins

C'est un algorithme simple qui va établir des prévisions en point x à partir des observations qui sont proches de x. Plus précisément, étant donné $k \leq n$ il est défini par

$$g_{n,k}(x) = \begin{cases} 1 & \text{si } \sum_{i \in \text{kppv}(x)} \mathbf{1}_{y_i=1} \geq \sum_{i \in \text{kppv}(x)} \mathbf{1}_{y_i=0} \\ 0 & \text{sinon.} \end{cases}$$

pour la prévision des groupes et par

$$S_{n,k}(x) = \frac{1}{|\text{kppv}(x)|} \sum_{i \in \text{kppv}(x)} \mathbf{1}_{y_i=1}$$

pour la prévision de la probabilité $\mathbf{P}(Y = 1|X = x)$ avec

$$\text{kppv}(x) = \{i \leq n : \|x - x_i\| \leq \|x - x_{(k)}\|\}$$

et $\|x - x_{(k)}\|$ la k^e plus petite valeur parmi $\{\|x - x_1\|, \ldots, \|x - x_n\|\}$.

Exercice 1.1 (kppv avec kknn)**.**

1. Séparer les données en un échantillon d'apprentissage de taille 300 et un échantillon test de taille 200.
2. A l'aide de la fonction **kknn** du package **kknn** entraîner la règle des 3 plus proches voisins sur les données d'apprentissage et calculer les groupes prédits par cette règle des données test.
3. En déduire une estimation de l'erreur de classification de la règle des 3 plus proches voisins.
4. Retrouver le résultat précédent à l'aide la fonction `accuracy` du package `yardstick`.
5. Toujours à l'aide des fonctions de `yardstick`, calculer la spécificité, le sensibilité et le balanced accuracy. On pourra consulter la page https://yardstick.tidymodels.org/articles/metric-types.html#metrics
6. Retrouver les résultats précédents en utilisant la fonction `metric_set`.

1.2 La courbe ROC

C'est un critère fréquemment utilisé pour mesurer la performance d'un **score**. Etant donné (X, Y) un couple aléatoire à valeurs dans $\mathcal{X} \times \{-1, 1\}$, on rappelle qu'un score est une fonction $S : \mathcal{X} \to \mathbb{R}$. Dans la plupart des cas, un score s'obtient en estimant la probabilité $\mathbf{P}(Y = 1|X = x)$. Pour un seuil $s \in \mathbb{R}$ fixé, un score possède deux types d'erreur

$$\alpha(s) = \mathbf{P}(S(X) \geq s|Y = -1) \quad \text{et} \quad \beta(s) = \mathbf{P}(S(X) < s|Y = 1).$$

La courbe ROC est la **courbe paramétrée** définie par :

$$\begin{cases} x(s) = \alpha(s) = \mathbf{P}(S(X) > s|Y = -1) \\ y(s) = 1 - \beta(s) = \mathbf{P}(S(X) \geq s|Y = 1) \end{cases}$$

Elle permet donc de visualiser sur une seul graphe 2D ces deux erreurs pour toutes les valeurs de seuil s.

Exercice 1.2 (Étude de la courbe ROC)**.** On considère dans cet exercice une fonction de score S que l'on suppose absolument continue.

1. Montrer que la courbe ROC vit dans le carré $[0, 1]^2$.

2. On suppose dans cette question que S est **parfait**, ce qui revient à dire qu'il sépare parfaitement les 2 groupes. Mathématiquement on traduit cela par l'existence d'un seuil $s^\star \in \mathbb{R}$ tel que

$$\mathbf{P}(Y = 1 | S(X) \geq s^\star) = 1 \quad \text{et} \quad \mathbf{P}(Y = -1 | S(X) < s^\star) = 1.$$

   Analyser la courbe ROC du score parfait.

3. On suppose dans cette question que S est **aléatoire** dans le sens où $S(X)$ est indépendante de Y (cela revient à dire que les notes $S(X)$ n'ont aucun lien avec le groupe). Analyser la courbe ROC d'un tel score.

Exercice 1.3 (Courbe ROC avec yardstick)**.** On reprend les données de l'exercice 1.1 et on considère toujours la règle des 3 plus proches voisins entrainée sur les données d'apprentissage.

1. Toujours pour la règle des 3 plus proches voisins, calculer les probabilités $\mathbf{P}(Y = 1 | X = x)$ prédites pour les données tests.

2. En déduire une estimation de la courbe ROC de la règle des 3 plus proches voisins. On pourra utiliser la fonction **roc_curve** du package **yardstick**.

Exercice 1.4 (Autres outils R pour la courbe ROC)**.** On dispose de 4 fonctions de score $S_j(x)$ dont on souhaite visualiser les courbes ROC à partir des valeurs de score calculés sur un échantillon. On trouvera dans le tableau `df` les scores $S_j(X_i), i = 1, \ldots, n$ ainsi que les observations des groupes Y_i

```
set.seed(12345)
n <- 200
Y <- rbinom(n,1,0.5)
S1 <- runif(n)
S2 <- S1
S2[Y==1] <- runif(sum(Y==1),0.6,1)
S2[Y==0] <- runif(sum(Y==0),0,0.6)
S3 <- S2
S3[Y==1][1:10] <- runif(10,0,0.6)
S3[Y==0][1:10] <- runif(10,0.6,1)
(tbl <- tibble(S1,S2,S3,Y=Y))
## # A tibble: 200 x 4
##        S1    S2      S3     Y
##     <dbl> <dbl>   <dbl> <int>
##  1 0.589  0.630 0.420       1
##  2 0.893  0.790 0.00624     1
##  3 0.124  0.706 0.302       1
##  4 0.513  0.692 0.439       1
##  5 0.664  0.203 0.806       0
##  6 0.766  0.404 0.666       0
##  7 0.0926 0.194 0.954       0
```

```
##  8 0.0676 0.838 0.198       1
##  9 0.556  0.664 0.492       1
## 10 0.651  0.942 0.531       1
## # i 190 more rows
```

1. Visualiser, pour chaque score, les valeurs de score en fonction de Y. Commenter
2. Visualiser sur un même graphe les trois courbes ROC. On pourra utiliser d'abord utiliser la fonction **roc** du package **pROC** puis la fonction **geom_roc** du **plotROC** et enfin la fonction **roc_curve** du package **yardstick**.
3. Calculer les `AUC` à l'aide de la fonction `aucde` `pROC` puis `roc_auc` de `yardstick`.
4. On rappelle que l'AUC vérifier la propriété suivante : si (X_1, Y_1) et (X_2, Y_2) sont indépendantes et de même loi que (X, Y) et que $S(X)$ est continue, on a

$$AUC(S) = \mathbf{P}(S(X_1) \geq S(X_2)|(Y_1, Y_2) = (1, -1)).$$

   Utiliser cette propriété pour retrouver l'AUC de S3.

1.3 Exercices complémentaires

Exercice 1.5 (Optimalité de la fonction de régression)**.** Soit (X, Y) un couple aléatoire à valeurs dans $\mathbb{R}^d \times \mathbb{R}$.

1. Rappeler la définition du risque quadratique en régression.
2. Montrer que la fonction de régression $m^\star(x) = \mathbf{E}[Y|X = x]$ minimise ce risque.

Exercice 1.6 (Optimalité de la règle de Bayes)**.** Soit (X, Y) un couple aléatoire à valeurs dans $\mathbb{R}^d \times \{0, 1\}$.

1. Rappeler la définition d'une fonction de prévision.
2. Rappeler la définition de la règle de Bayes $g^\star$ et de l'erreur de Bayes $L^\star$.
3. Soit g une fonction de décision. Montrer que pour tout $x \in \mathbb{R}^d$ on a

$$\mathbf{P}(g(X) \neq Y|X = x) = 1 - (\mathbf{1}_{g(x)=1}\eta(x) + \mathbf{1}_{g(x)=0}(1 - \eta(x)))$$

   où $\eta(x) = \mathbf{P}(Y = 1|X = x)$.
4. En déduire que pour tout $x \in \mathbb{R}^d$ et pour toute fonction de prévision g

$$\mathbf{P}(g(X) \neq Y|X = x) - \mathbf{P}(g^\star(X) \neq Y|X = x) \geq 0.$$

   Conclure.

5. On considère (X, Y) un couple aléatoire à valeurs dans $\mathbb{R} \times \{0, 1\}$ tel que

$$X \sim \mathcal{U}[-2, 2] \quad \text{et} \quad (Y|X = x) \sim \begin{cases} \mathcal{B}(1/5) & \text{si } x \leq 0 \\ \mathcal{B}(9/10) & \text{si } x > 0 \end{cases}$$

où $\mathcal{U}[a, b]$ désigne la loi uniforme sur $[a, b]$ et $\mathcal{B}(p)$ la loi de Bernoulli de paramètre p. Calculer la règle de Bayes et l'erreur de Bayes.

2 Estimation du risque par ré-échantillonnage

Rappelons que, pour une fonction de perte ℓ donnée, le risque de f_n est défini par

$$\mathcal{R}(f_n) = \mathbf{E}[\ell(Y, f_n(X, \mathcal{D}_n))] = \mathbf{E}_{\mathcal{D}_n}[\mathbf{E}_{(X,Y)}[\ell(Y, f_n(X, \mathcal{D}_n))]].$$

Il représente la perte moyenne de f_n par rapport aux lois de (X, Y) et de $\mathcal{D}_n$. Ces lois étant inconnues, l'utilisateur n'a jamais accès à $\mathcal{R}(f_n)$ et doit donc l'estimer. Nous présentons dans ce chapitre les méthodes classiques qui reposent sur des techniques de ré-échantillonnage.

2.1 Validation croisée

Exercice 2.1 (Calculs de blocs avec rsample)**.** Le package **rsample** permet d'obtenir des partitions des données pour calculer des estimations de risques. On s'intéresse dans cet exercice aux fonctions

- **mc_cv** pour partager les données en 2 (*validation hold out*)
- **vfold_cv** pour partager les données en blocs (*validation croisée*)

On reprend les données générées avec la fonction du chapitre précédent :

```
donnees <- gen_class_bin2D(n=500,graine=12345,bayes=0.20)$donnees
```

1. À l'aide de **mc_cv** séparer les données en 2 échantillons apprentissage/test avec 2/3 des données dans l'échantillon d'apprentissage.
2. Afficher l'index des individus de l'échantillon d'apprentissage. Faire de même pour l'échantillon test.
3. À l'aide de **vfold_cv** séparer les données en 10 blocs.
4. Afficher l'index des individus qui se trouvent dans le bloc 4.

On considère la perte indicatrice : $\ell(y, y') = \mathbf{1}_{y \neq y'}$, le risque d'un algorithme f est donc

$$\mathcal{R}(f) = \mathbf{E}[\mathbf{1}_{Y \neq f(X)}] = \mathbf{P}(Y \neq f(X)),$$

il est appelé **probabilité d'erreur** ou **erreur de classification**. On s'intéresse à l'estimation de ce risque par des méthodes de ré-échantillonnage pour l'algorithme des k plus proches voisins.

Exercice 2.2 (Progammation de validation hold out et validation croisée)**.**

1. Créer une fonction qui admet en entrée

   - un jeu de données
   - une valeur de `k`
   - une séparation des données en apprentissage/test issue de `mc_cv`

   et renvoie en sortie l'erreur de classification de la règle des k plus proches voisins calculées sur l'échantillon test. On pourra la tester pour `k=3`.

2. Refaire la question précédente avec une validation croisée K blocs.

3. On considère la grille de valeurs de `k`

   ```
   grille.k <- c(1:100)
   ```

   Sélectionner dans cette grille une valeur de k en minimisant l'erreur de classification calculée par validation hold out, puis par validation croisée 10 blocs. On pourra également visualiser les courbes de risque (erreur de classification en fonction du nombre de voisins).

2.2 Le package tidymodels

Il s'agit d'un meta-package qui regroupe plusieurs packages (`parsnip`, `recipes`, `rsample`...) qui utilisent une approche *tidy* pour **sélectionner un algorithme de prévision**. Il propose de créer un projet de travail, appelé **workflow**, permettant de construire un algorithme de prévision. Cette construction requiert tout un ensemble d'étapes comme la gestion des données manquantes, la création ou suppression de variables explicatives, la recherche de colinéarité parmi les variables explicatives, la sélection des paramètres de l'algorithme... Toutes ces étapes s'enchaînent en mettant à jour le workflow. Nous ne présentons pas toutes les fonctionnalités de `tidymodels`, on pourra les retrouver dans le tutoriel présentant le package à l'url https://www.tidymodels.org. Nous nous concentrons uniquement sur le choix de paramètres d'algorithmes de prévision à travers l'exemple des plus proches voisins.

La première étape est consacrée à la définition de l'algorithme. Pour des plus proches voisins en classification on utilise

```
library(tidymodels)
tune_spec <-
  nearest_neighbor(neighbors=tune(),weight_func="rectangular") |>
  set_mode("classification") |>
  set_engine("kknn")
```

Dans la fonction `nearest_neighbor` nous précisions que nous souhaitons utiliser le noyau `rectangular` comme nous l'avons fait jusqu'à présent. L'option `neighbors=tune()` spécifie que le nombre de plus proches voisins est un paramètre à calibrer. Plusieurs packages `R` permettent d'entraîner des k plus proches voisins, on utilise `set_engine("kknn")` pour indiquer que l'on souhaite utiliser la fonction `kknn`. On pourra trouver l'ensemble des algorithmes disponibles dans `tidymodels` ainsi que les noms de leurs paramètres ici : https://www.tidymodels.org/find/parsnip/.

On initialise le projet de travail dans la seconde étape en créant le *workflow* :

```
ppv_wf <- workflow() |>
  add_model(tune_spec) |>
  add_formula(Y ~ .)
```

On renseigne dans `add_model` les différentes informations sur l'algorithme des plus proches voisins et on indique dans `add_formula` la variable à expliquer et les variables explicatives. Nous avons vu dans la section précédente que la recherche du meilleur nombre de voisins s'effectue en estimant un risque de prévision sur une grille de valeurs candidates. On considère toujours des valeurs de k variant entre 1 et 100 :

```
grille_k <- tibble(neighbors=1:100)
```

Le choix de k s'effectue avec la fonction `tune_grid` du package `tune` qui possède notamment les arguments suivants :

```
tune_grid(...,resamples=...,grid=...,metrics=...)
```

On indique dans :

- `resamples` le ré-échantillonnage. Par exemple les échantillons d'apprentissage et test pour la validation hold out (avec la fonction `mc_cv`), les blocs pour la validation croisée (`vfolds_cv`), les tirages bootstraps (`bootstraps`) ;
- `grid` la grille de paramètres candidats ;
- `metrics` le risque de prévision (erreur quadratique, erreur de classification, AUC...).

Exercice 2.3 (Validation hold out et validation croisée avec tidymodels)**.**

1. Retrouver les erreurs par validation hold out en utilisant `tune_grid` et `collect_metrics`.
2. Visualiser les meilleures valeurs de `k` à l'aide de `show_best`

3. Sélectionner la meilleur valeur de `k` avec `select_best`.
4. Finaliser l'algorithme à l'aide de `finalize_workflow` et `fit`.
5. Faire le même travail pour la validation croisée 10 blocs.

Exercice 2.4 (Validation croisée pour l'AUC)**.** Sélectionner la valeur de k qui maximise l'AUC calculé par validation croisée 10 blocs.

3 Compléments

3.1 Variance de l'erreur de validation croisée

Lorsque les estimateurs du risque sont obtenus en ajustant l'algorithme plusieurs fois (sur différent sous-échantillons), il est possible d'estimer la variance (conditionnelle aux données) de ces estimateurs. Par exemple, dans le cas d'une validation croisée K blocs pour le risque quadratique en régression, cette variance est approchée par

$$\frac{1}{K}\frac{1}{K-1}\sum_{k=1}^{K}(\mathcal{R}_k - \mathcal{R}_{\mathrm{CV}})^2$$

où $\mathcal{R}_k$ représente l'erreur sur le bloc k et $\mathcal{R}_{\mathrm{CV}}$ est l'erreur de validation croisée. L'estimation de cette variance est très importante puisqu'elle permet d'avoir une idée sur les erreurs obtenues. Elle peut donc aider l'utilisateur à calibrer les algorithmes.

Exercice 3.1 (Variance de l'estimateur du risque)**.** On reprend les données générées avec la fonction des chapitres précédents :

```
donnees <- gen_class_bin2D(n=500,graine=12345,bayes=0.20)$donnees
```

et on considère les estimateurs du risque obtenus avec la validation croisée suivants :

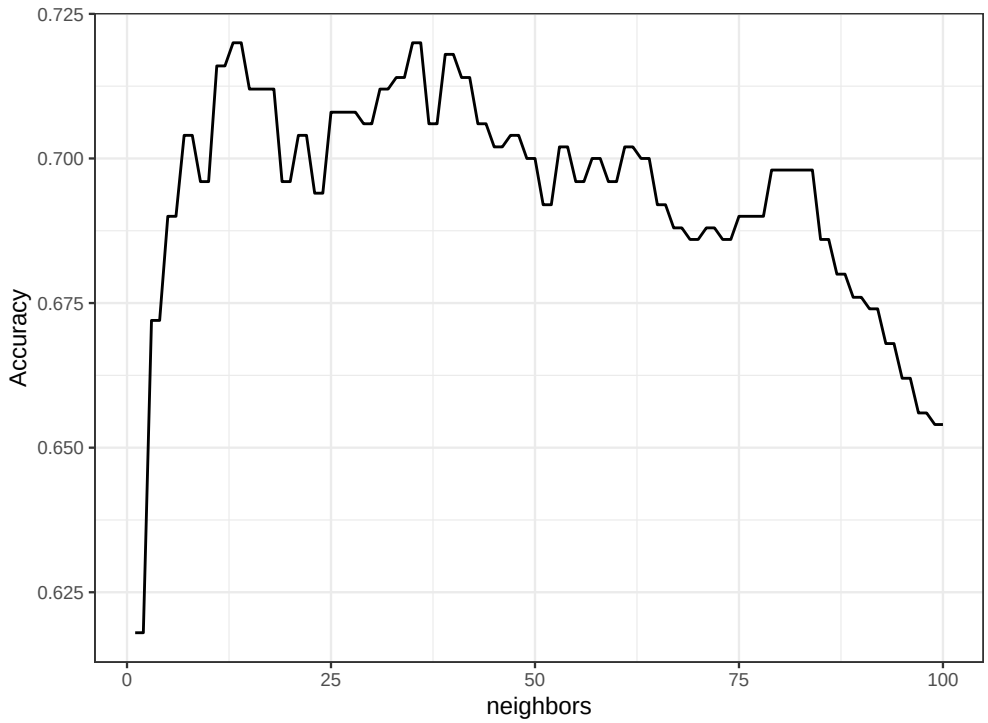
```
set.seed(12345)
tune_spec <-
  nearest_neighbor(neighbors=tune(),weight_func="rectangular") |>
  set_mode("classification") |>
  set_engine("kknn")
ppv_wf <- workflow() |>
  add_model(tune_spec) |>
  add_formula(Y ~ .)

grille_k <- tibble(neighbors=1:100)
re_ech_cv <- vfold_cv(donnees,v=10)

ppv.cv <- ppv_wf |>
  tune_grid(
    resamples = re_ech_cv,
    grid = grille_k,
    metrics=metric_set(accuracy))
```

On visualise la courbe de risque avec

```
tbl <- ppv.cv |> collect_metrics()
ggplot(tbl)+aes(x=neighbors,y=mean)+geom_line()+ylab("Accuracy")
```

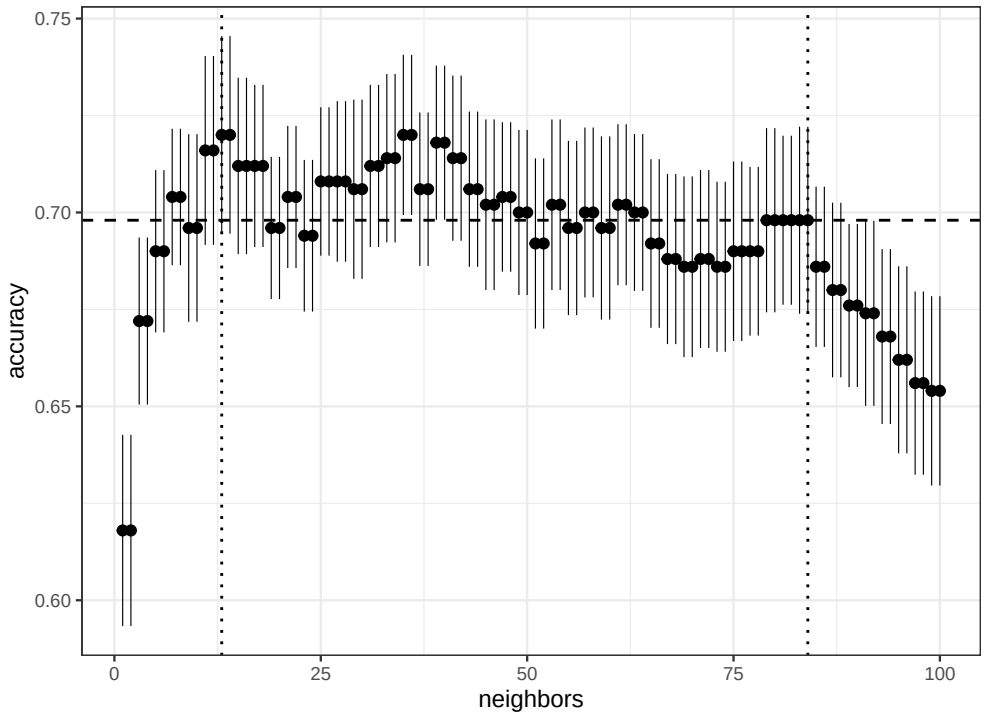



```
(res.cv <- ppv.cv |> collect_metrics())
## # A tibble: 100 x 7
##    neighbors .metric  .estimator  mean     n std_err .config
##        <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
##  1         1 accuracy binary     0.618    10  0.0247 Prepro~
##  2         2 accuracy binary     0.618    10  0.0247 Prepro~
##  3         3 accuracy binary     0.672    10  0.0215 Prepro~
##  4         4 accuracy binary     0.672    10  0.0215 Prepro~
##  5         5 accuracy binary     0.69     10  0.0209 Prepro~
##  6         6 accuracy binary     0.69     10  0.0209 Prepro~
##  7         7 accuracy binary     0.704    10  0.0176 Prepro~
##  8         8 accuracy binary     0.704    10  0.0176 Prepro~
##  9         9 accuracy binary     0.696    10  0.0242 Prepro~
## 10        10 accuracy binary     0.696    10  0.0242 Prepro~
## # i 90 more rows
```

1. Expliquer à quoi correspond la colonne `std_err` du tibble `tbl`.
2. Expliquer la sortie de

```
(k_one_std_err <- ppv.cv |>
   select_by_one_std_err(desc(neighbors),metric="accuracy"))
## # A tibble: 1 x 2
##   neighbors .config
##       <int> <chr>
## 1        84 Preprocessor1_Model084
(err_one_std_err <- res.cv |>
    filter(neighbors==k_one_std_err$neighbors))
## # A tibble: 1 x 7
##   neighbors .metric  .estimator  mean     n std_err .config
##       <int> <chr>    <chr>      <dbl> <int>   <dbl> <chr>
## 1        84 accuracy binary     0.698    10  0.0241 Preproc~
```

3. Visualiser les écart-types estimés sur la courbe de risque ainsi que la valeur de k optimale et celle de la question précédente. On pourra par exemple tracer le graphe suivant :



3.2 Répéter les méthodes de ré-échantillonnage

Les méthodes d'estimation du risque présentées dans cette partie (hold out, validation croisée) sont basées sur du **ré-échantillonnage**. Elles peuvent se révéler sensible à la manière de couper l'échantillon. C'est pourquoi il est recommandé de les **répéter plusieurs fois** et de moyenner les erreurs sur les répétitions. Ces répétitions sont très faciles à mettre en œuvre avec l'approche `tidymodels`, il suffit de construire les échantillons adéquat. Par exemple,

- pour répéter 20 fois une validation hold out, on définit les blocs avec

```
ho_rep <- mc_cv(donnees,prop=2/3,times = 20)
```

- pour la validation croisée, on utilise

```
cv_rep <- vfold_cv(donnees, v = 10, repeats = 20)
```

Le reste ne change pas, il faudra utiliser ces objets dans l'option `resamples` de `tune_grid`.

Exercice 3.2 (Validation croisée répétée)**.** Refaire l'exercice 3.1 avec une validation croisée répétée 20 fois. On comparera notamment la variance des estimateurs obtenus.

3.3 Calcul parallèle

Les validations croisées (répétées) peuvent se révéler coûteuses en temps de calcul. On utilise souvent des techniques de parallélisation pour améliorer les performances computationnelles. Ces techniques sont relativement facile à mettre en œuvre avec `tunegrid`, on peut par exemple utiliser la librairie **doParallel** pour utiliser plusieurs cœurs de la machine. À titre d'illustration, nous comparons ci-dessous les temps de calcul pour faire une validation croisée répétée 20 fois. On commence sans calcul parallèle

```
grille_k1 <- tibble(neighbors=c(1,11,21,31))
t1 <- system.time(
  ppv_wf |> tune_grid(
  resamples = cv_rep,
  grid = grille_k1))
```

```
t1
##    user  system elapsed
## 146.115   5.331 152.338
```

Pour paralléliser on regarde tout d'abord la nombre de cœurs disponibles sur la machine

```
library(doParallel)
detectCores()
## [1] 8
```

On ouvre les connexions sur chaque cœur

```
cl <- makePSOCKcluster(4)
registerDoParallel(cl)
```

et on lance la validation croisée, elle sera automatiquement parallélisée

```
t2 <- system.time(
  ppv_wf |> tune_grid(
  resamples = cv_rep,
  grid = grille_k1))
```

```
t2
##    user  system elapsed
##   0.490   0.094  32.988
```

Le temps de calcul a bien été diminué. Enfin on n'oublie pas de refermer les connexions

```
stopCluster(cl)
```

partie II

Algorithmes linéraires

4 Analyse discriminante linéaire

L'analyse discriminante linéaire est un algorithme de référence en classification supervisée. Il peut être appréhendé de deux façons complémentaires :

- une approche **géométrique** qui revient à chercher des hyperplans qui séparent au mieux les groupes ;
- une approche **modèle** qui fait l'hypothèse que les lois des covariables sont des vecteurs gaussiens avec des valeurs de paramètres différentes pour chaque groupe.

On considère $(x_1, y_1), \ldots, (x_n, y_n)$ un échantillon où x_i est à valeurs dans $\mathbb{R}^d$ et y_i dans $\{0, 1\}$. L'approche **géométrique** revient à chercher une droite de $\mathbb{R}^d$ d'équation $a_1 x_1 + \cdots + a_d x_d = 0$ telle que :

- les centres de gravité de chaque groupe projeté sur cette droite soit au mieux séparé $\Longrightarrow$ maximiser la distance inter-classe.
- les observations projetés soient proches de leur centre de gravité projeté $\Longrightarrow$ minimiser la distance intra-classe.

Le compromis entre ces deux distances s'obtient en maximisant le **coefficient de Rayleigh** qui est le quotient entre ces deux distance :

$$J(a) = \frac{B(a)}{W(a)} = \frac{a^t B a}{a^t W a}$$

où B et W sont les matrices inter et intra classes définies pas

$$B = \frac{1}{n} \sum_{k=1}^{K} n_k (g_k - g)(g_k - g)^t \quad \text{et} \quad W = \frac{1}{n} \sum_{k=1}^{K} n_k V_k \quad \text{avec} \quad V_k = \frac{1}{n_k} \sum_{i:Y_i=k} (X_i - g_k)(X_i - g_k)^t.$$

Ici g désigne le centre de gravité du nuage $x_i, i = 1, \ldots, n$ et $g_k, k = 0, 1$ les centres de gravité des deux groupes. La solution est donnée par un vecteur propre associé à la plus grande valeur propre de $W^{-1}B$.

L'approche **modèle** fait l'hypothèse que les vecteurs $X|Y = k, k = 0, 1$ sont des vecteurs gaussiens d'espérance $\mu_k \in \mathbb{R}^d$ et de matrice de variance covariance Σ. Ces paramètres sont estimés par maximum de vraisemblance et on déduit les probabilités a posteriori par la **formule de Bayes** :

$$\mathbf{P}(Y = k|X = x) = \frac{\pi_k f_{X|Y=k}(x)}{f(x)}.$$

Le lien entre ces deux approches est établi dans l'exercice @ref(exr:exo-calcul-axes-lda). Nous proposons dans cette partie quelques exercices pour mettre en œuvre et analyser des analyses discriminantes avec **R**.

4.1 Prise en main : LDA et QDA sur les iris de Fisher

On considère les données sur les iris de Fisher.

```
data(iris)
```

1. A l'aide de la fonction **PCA** du package **FactoMineR**, réaliser une ACP en utilisant comme variables actives les 4 variables quantitatives du jeu de données. On mettra la variable `Species` comme variable qualitative supplémentaire (option `quali.sup`).
2. Représenter le nuage des individus sur les 2 premiers axes de l'ACP en utilisant une couleur différente pour chaque espèce d'iris (option `habillage`).
3. A l'aide de la fonction **lda** du package **MASS**, effectuer une analyse discriminante linéaire permettant d'expliquer l'espèce par les 4 autres variables explicatives.
4. Représenter le nuage des individus sur les deux premiers axes de l'analyse discriminante linéaire (en utilisant une couleur différente pour chaque espèce d'iris).
5. Rappeler comment sont obtenues les coordonnées des individus sur chaque axe. En déduire une interprétation de la position des individus.
6. Comparer les représentations des questions 2 et 4.
7. Expliquer les sorties des commandes suivantes (`mod.lda` est l'objet construit avec la fonction **lda**).

   ```
   score <- predict(mod.lda)$x
   ldahist(score[,1],iris[,5])
   ldahist(score[,2],iris[,5])
   ```

8. Exécuter et analyser les sorties de la commande

   ```
   mod.lda2 <- lda(Species~.,data=iris,CV=TRUE)
   ```

9. Comparer, en terme d'erreur de prévision, les performances de LDA et QDA.

4.2 Un exemple multi-classes

On considère les jeux de données **Vowel** (training et test) qui se trouvent à cet url. On peut les importer avec

```
dapp <- read_csv("https://web.stanford.edu/~hastie/ElemStatLearn/datasets/vowel.train")[,-1]
dtest <- read_csv("https://web.stanford.edu/~hastie/ElemStatLearn/datasets/vowel.test")[,-1]
```

1. Expliquer le problème.
2. Effectuer une analyse discriminante linéaire (uniquement avec les données d'apprentissage) et visualiser les individus sur les 2 premiers axes de l'analyse discriminante. On pourra utiliser **predict**.
3. La fonction suivante permet de choisir les axes à visualiser, ainsi que les centres de gravité projetés des groupes.

   ```
   repres_axes <- function(prev,cdg,axe1=1,axe2=2){
     cdg <- prev %>% group_by(y) %>% summarise_all(mean)
     nom1 <- paste("LD",as.character(axe1),sep="")
     nom2 <- paste("LD",as.character(axe2),sep="")
     ggplot(prev)+aes_string(x=as.name(nom1),y=as.name(nom2))+
   geom_point(aes(color=y))+
   geom_point(data=cdg,aes(color=y),shape=17,size=4)+
   theme_classic()
   }
   ```

   Étudier la pertinence des axes.
4. Représenter les individus sur le premier plan factoriel de l'ACP, on utilisera une couleur différente pour chaque groupe. On pourra utiliser le package **FactoMineR**.
5. Comparer cette projection avec celle obtenue par l'analyse discriminante linéaire.
6. Évaluer la performance de la **lda** sur les données test. Comparer avec l'analyse discriminante quadratique.
7. Expliquer comment on peut faire de la prévision en réduisant la dimension de l'espace des X.
8. Proposer une méthode permettant de choisir le meilleur nombre d'axes. On pourra notamment utiliser l'option `dimen` de la fonction **predict.lda**.

4.3 Grande dimension : reconnaissance de phonèmes

On considère le jeu de données **phoneme** téléchargeable à l'url https://github.com/cran/ElemStatLearn/blob/master/data/phoneme.RData.

```
load('data/phoneme.RData')
data(phoneme)
donnees <- phoneme[,-c(258)]
```

1. Expliquer le problème et représenter pour chaque groupe la courbe moyenne.
2. Séparer les données en un échantillon d'apprentissage de taille 3000 et un échantillon test de taille 1509.
3. Effectuer une analyse discriminante linéaire et une analyse discriminante quadratique sur les données d'apprentissage uniquement. Évaluer les performances de ces deux approches sur les données test.
4. Quels peuvent être les intérêts d'effectuer une analyse discriminante régularisée dans ce contexte ? Effectuer une telle analyse à l'aide de la fonction **rda** du package **klaR**.
5. Sélectionner les paramètres de régularisation à l'aide du package **caret**. Comparer le nouveau modèle aux précédents.

4.4 Exercices

Exercice 4.1 (MV pour LDA)**.** On cherche à expliquer une variable aléatoire Y à valeurs dans $\{0,1\}$ par une variable aléatoire X à valeurs dans $\mathbb{R}$.

1. Quels sont les paramètres à estimer dans le modèle d'analyse discriminante linéaire.
2. Calculer la vraisemblance conditionnelle à Y et en déduire les estimateurs des paramètres des lois gaussiennes.
3. Comparer les estimateurs obtenus avec ceux du cours.

Exercice 4.2 (Fonctions linéaires discriminantes)**.** On cherche à expliquer une variable aléatoire Y à valeurs dans $\{0,1\}$ par une variable aléatoire X à valeurs dans $\mathbb{R}^p$.

1. Rappeler le modèle d'analyse discriminante linéaire.
2. Soit $x \in \mathbb{R}^p$ un nouvel individu. Montrer que la règle qui consiste à affecter x dans le groupe qui maximise $\mathbf{P}(Y = k|X = x)$ est équivalente à la règle qui consiste à affecter x dans le groupe qui maximise les fonctions linéaires discriminantes (on prendra soin de rappeler la définition des fonctions linéaires discriminantes).

Exercice 4.3 (Approche géométrique de la LDA)**.** On considère un n-échantillon i.i.d. $(x_1, y_1), \ldots, (x_n, y_n)$ où x_i est à valeurs dans $\mathbb{R}^2$ et y_i dans $\{0, 1\}$. On cherche une droite vectorielle a telle que les projections de chaque groupe sur a soient séparées "au mieux". Dit autrement, on cherche a telle que

- la distance entre les centres de gravité

$$g_0 = \frac{1}{\text{card}\{i : y_i = 0\}} \sum_{i:y_i=0} x_i \quad \text{et} \quad g_1 = \frac{1}{\text{card}\{i : y_i = 1\}} \sum_{i:y_i=1} x_i$$

  projetés sur a soit maximale (cette distance est appelée distance interclasse) ;
- la distance entre les projections des individus et leur centre de gravité soit minimale (distance interclasse).

Pour un vecteur u de $\mathbb{R}^2$, on désigne par $\pi_a(u)$ son projeté sur la droite engendrée par a. Sans perte de généralité on supposera dans un premier temps que a est de norme 1.

1. Rappeler les définitions des variances totale V, intra W et inter B des observations $(x_1, y_1), \ldots, (x_n, y_n)$.
2. Pour u fixé dans $\mathbb{R}^2$, exprimer $\pi_a(u)$ en fonction de u et a et en déduire que $\|\pi_a(u)\|^2 = a^t u u^t a$.
3. Exprimer les variances totale $V(a)$, intra $W(a)$ et inter $B(a)$ projetées sur a en fonction des variances calculées à la question 1.
4. On cherche maintenant à maximiser

$$J(a) = \frac{B(a)}{W(a)}$$

   ou encore à

$$\text{maximiser } B(a) \quad \text{sous la contrainte} \quad W(a) = 1. \tag{4.1}$$

   La méthode des multiplicateurs de Lagrange permet de résoudre un tel problème. La solution du problème de maximisation d'une fonction $f(x)$ sujette à $h(x) = 0$ s'obtient en résolvant l'équation

$$\frac{\partial L(x, \lambda)}{\partial x} = 0, \quad \text{où} \quad L(x, \lambda) = f(x) + \lambda h(x).$$

   a. Montrer que la solution du problème équation 4.1 est un vecteur propre de $W^{-1}B$ associé à la plus grande valeur propre de $W^{-1}B$. On note $a^\star$ cette solution.

b. Montrer que $a^\star$ est colinéaire à $W^{-1}(g_1 - g_0)$. On pourra admettre que, dans le cas de 2 groupes, on a

$$B = \frac{n_0 n_1}{n^2}(g_1 - g_0)(g_1 - g_0)^t.$$

c. On considère la règle géométrique d'affectation qui consiste à classer un nouvel individu $x \in \mathbb{R}^p$ au groupe 1 si son projeté sur $a^\star$ est plus proche de $\pi_{a^\star}(g_1)$ que de $\pi_{a^\star}(g_0)$. Montrer que x sera affecté au groupe 1 si

$$S(x) = x^t W^{-1}(g_1 - g_0) > s$$

où on exprimera s en fonction de g_0, g_1 et W.

d. Montrer que cette règle est équivalente à choisir le groupe qui minimise la distance de Mahalanobis

$$d(x, g_k) = (x - g_k)^t W^{-1}(x - g_k), \quad k = 0, 1.$$

e. On revient maintenant à l'approche probabiliste de l'analyse discriminante linéaire vue en cours et on considère la règle d'affectation qui consiste à décider "groupe 1'' si $\mathbf{P}(Y = 1|X = x) \geq 0.5$. Montrer que dans ce cas, un nouvel individu x est affecter au groupe 1 si :

$$S(x) = x^t \Sigma^{-1}(\mu_1 - \mu_0) > \frac{1}{2}(\mu_1 + \mu_0)^t \Sigma^{-1}(\mu_1 - \mu_0) - \log\left(\frac{\pi_1}{\pi_0}\right).$$

Conclure.

5 Support Vector Machine (SVM)

Etant donnée un échantillon $(x_1, y_1), \ldots, (x_n, y_n)$ où les x_i sont à valeurs dans $\mathbb{R}^p$ et les y_i sont binaires à valeurs dans $\{-1, 1\}$, l'approche **SVM** cherche le **meilleur hyperplan** en terme de séparation des données. Globalement on veut que les `1` se trouvent d'un coté de l'hyperplan et les `-1` de l'autre. Dans cette partie on propose d'étudier la mise en œuvre de cet algorithme tout d'abord dans le cas idéal où les données sont séparables puis dans le cas plus réel où elles ne le sont pas. Nous verrons ensuite comment introduire de la non linéarité ne utilisant l'**astuce du noyau**.

5.1 Cas séparable

Le cas séparable est le cas facile : il correspond à la situation où il existe effectivement un (même plusieurs) hyperplan(s) qui sépare(nt) parfaitement les 1 des -1. Il ne se produit quasiment jamais en pratique mais il convient de l'étudier pour comprendre comment est construit l'algorithme. Dans ce cas on cherche l'hyperplan d'équation $\langle w, x\rangle + b = w^t x + b = 0$ tel que la **marge** (qui peut être vue comme la distance entre les observations les plus proches de l'hyperplan et l'hyperplan) soit maximale. Mathématiquement le problème se réécrit comme un problème d'optimisation sous contraintes :

$$\min_{w,b} \frac{1}{2}\|w\|^2 \tag{5.1}$$

$$\text{sous les contraintes } y_i(w^t x_i + b) \geq 1,\ i = 1, \ldots, n.$$

La solution s'obtient de façon classique en résolvant le problème dual et elle s'écrit comme une combinaison linéaire des x_i

$$w^\star = \sum_{i=1}^{n} \alpha_i^\star y_i x_i.$$

De plus, les conditions **KKT** impliquent que pour tout $i = 1, \ldots, n$:

- $\alpha_i^\star = 0$

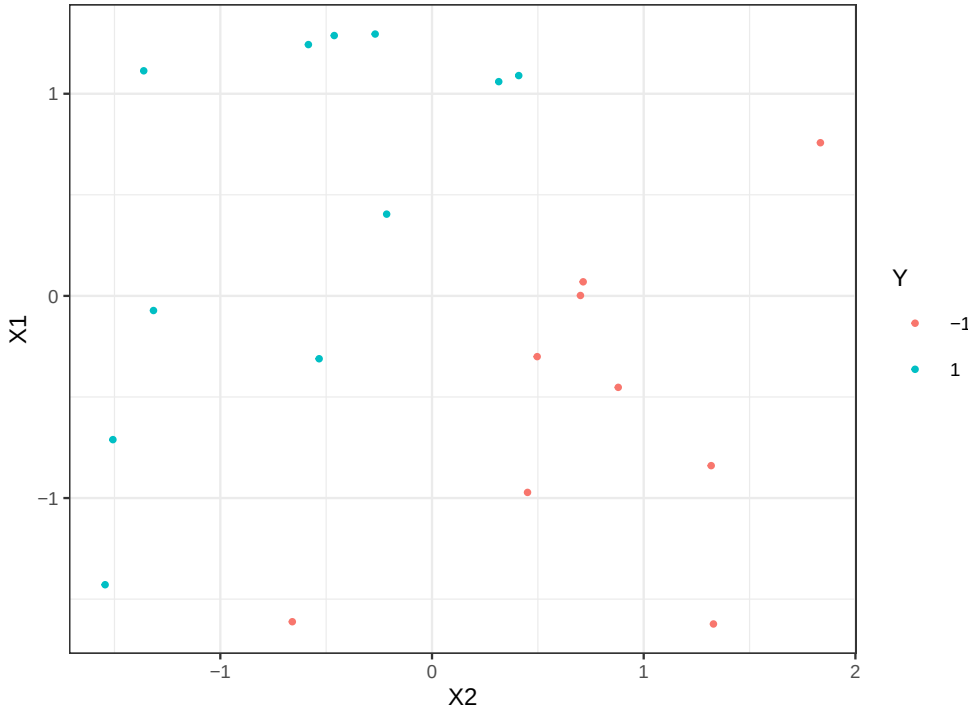
ou

- $y_i(x_i^t w + b) - 1 = 0$.

Ces conditions impliquent que $w^\star$ s'écrit comme une combinaison linéaire de quelques points, appelés **vecteurs supports** qui se trouvent **sur la marge**. Nous proposons dans l'exercice suivant de retrouver ces points et de tracer la marge sur un exemple simple.

Exercice 5.1 (SVM - cas séparable)**.** On considère le nuage de points suivant :

```
n <- 20
set.seed(123)
X1 <- scale(runif(n))[,1]
set.seed(567)
X2 <- scale(runif(n))[,1]
Y <- rep(-1,n)
Y[X1>X2] <- 1
Y <- as.factor(Y)
donnees <- tibble(X1,X2,Y)
(p <- ggplot(donnees)+aes(x=X2,y=X1,color=Y)+geom_point())
```



La fonction **svm** du package **e1071** permet d'ajuster une SVM :

```
library(e1071)
mod.svm <- svm(Y~.,data=donnees,kernel="linear",cost=10000000000)
```

1. Récupérer les vecteurs supports et visualiser les sur le graphe (en utilisant une autre couleur par exemple). On les affectera à un **tibble** dont les 2 premières colonnes représenteront les valeurs de X_1 et X_2 des vecteurs supports.

```
ind.svm <- mod.svm$index
sv <- donnees |> slice(ind.svm)
...
```

2. Représenter sur le graphe l'hyperplan séparateur.

```
##         X1         X2
## -1.745100  2.136029
## [1] -0.4035113
```

3. Retrouver ce graphe à l'aide de la fonction **plot**.
4. Rappeler la règle de décision associée à la méthode SVM. Donner les estimations des paramètres de la règle de décision sur cet exemple. On pourra notamment regarder la sortie `coef` de la fonction **svm**.
5. On dispose d'un nouvel individu $x = (-0.5, 0.5)$. Expliquer comment on peut prédire son groupe.
6. Retrouver les résultats de la question précédente à l'aide de la fonction **predict**. On pourra utiliser l'option `decision.values = TRUE`.
7. Obtenir les probabilités prédites à l'aide de la fonction **predict**. On pourra utiliser `probability=TRUE` dans la fonction **svm**.

5.2 Cas non séparable

Dans la vraie vie, les groupes ne sont généralement pas séparables et il n'existe donc pas de solution au problème 5.1. On va donc autoriser certains points à être :

- mal classés

et/ou

- bien classés mais à l'intérieur de la marge.

Mathématiquement, cela revient à introduire des **variables ressorts** (**slacks variables**) $\xi_1, \ldots, \xi_n$ positives telles que :

- $\xi_i \in [0, 1] \Longrightarrow i$ bien classé mais **dans** la région définie par la **marge** ;
- $\xi_i > 1 \Longrightarrow i$ **mal classé**.

Le problème d'optimisation est alors de minimiser en (w, b, ξ)

$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i$$
$$\text{sous les contraintes } \begin{cases} y_i(w^t x_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0, i = 1, \ldots, n. \end{cases}$$

Le paramètre $C > 0$ est à **calibrer** et on remarque que le cas séparable correspond à $C \to +\infty$. Les solutions de ce nouveau problème d'optimisation s'obtiennent de la même façon que dans le cas séparable, en particulier $w^\star$ s'écrit toujours comme une combinaison linéaire

$$w^\star = \sum_{i=1}^{n}\alpha_i^\star y_i x_i.$$

de **vecteurs supports** sauf qu'on distingue deux types de vecteurs supports ($\alpha_i^\star > 0$):

- ceux **sur la frontière** définie par la marge : $\xi_i^\star = 0$;
- ceux **en dehors** : $\xi_i^\star > 0$ et $\alpha_i^\star = C$.

Le choix de C est crucial : ce paramètre régule le **compromis biais/variance** de la svm :

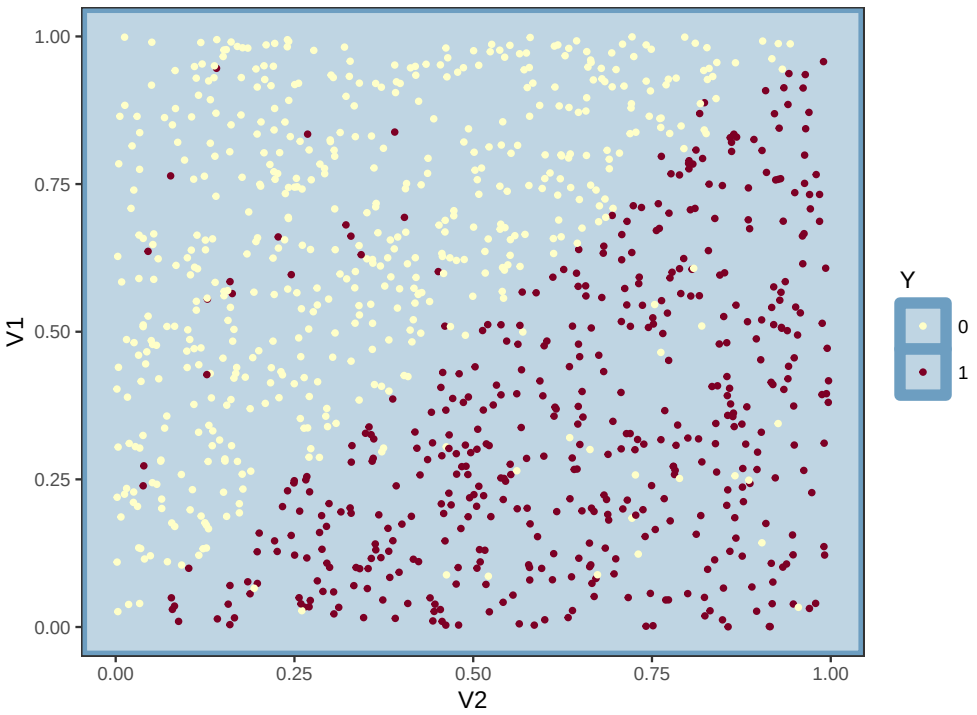
- $C \searrow$: la marge est privilégiée et les $\xi_i \nearrow \implies$ beaucoup d'observations dans la marge ou **mal classées** (et donc **beaucoup de vecteurs supports**).
- $C \nearrow$: $\xi_i \searrow$ donc moins d'observations mal classées $\implies$ **meilleur ajustement** mais petite marge $\implies$ risque de **surajustement**.

On choisit généralement ce paramètre à l'aide des techniques présentées dans le chapitre 2 :

- choix d'une grille de valeurs de C et d'un critère ;
- choix d'une méthode de ré-échantillonnage pour estimer le critère ;
- choix de la valeur de C qui minimise le critère estimé.

Exercice 5.2 (SVM - cas non séparable)**.** On considère le jeu de données `df3` définie ci-dessous.

```
n <- 1000
set.seed(1234)
df <- as.data.frame(matrix(runif(2*n),ncol=2))
df1 <- df |> filter(V1<=V2)|> mutate(Y=rbinom(n(),1,0.95))
df2 <- df |> filter(V1>V2)|> mutate(Y=rbinom(n(),1,0.05))
df3 <- bind_rows(df1,df2) |> mutate(Y=as.factor(Y))
ggplot(df3)+aes(x=V2,y=V1,color=Y)+geom_point()+
  scale_color_manual(values=c("#FFFFC8", "#7D0025"))+
  theme(panel.background = element_rect(fill = "#BFD5E3", colour = "#6D9EC1",linewidth = 2, linetype
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank())
```



1. Ajuster 3 **svm** en considérant comme valeur de C : 0.000001, 0.1 et 5. On pourra utiliser l'option `cost`.

```
mod.svm1 <- svm(Y~.,data=df3,kernel="linear",...)
mod.svm2 <- svm(Y~.,data=df3,kernel="linear",...)
mod.svm3 <- svm(Y~.,data=df3,kernel="linear",...)
```

2. Calculer les nombres de vecteurs supports pour chaque valeur de C.
3. Visualiser les 3 svm obtenues. Interpréter.
4. Sélectionner une valeur de C parmi les trois proposées. On pourra utiliser la fonction `tune_grid` de `tidymodels` avec

```
tune_spec <- svm_poly(cost=tune(),degree=1,scale_factor=1) |>
  set_mode("classification") |>
  set_engine("kernlab")
```

5.3 L'astuce du noyau

Les SVM présentées précédemment font l'hypothèse que les groupes sont **linéairement séparables**, ce qui n'est bien entendu pas toujours le cas en pratique. L'**astuce du noyau** permet de mettre de la non linéarité, elle consiste à :

- plonger les données dans un nouvel espace appelé **espace de représentation** ou **feature space** ;
- appliquer une **svm** linéaire dans ce nouvel espace.

Le terme **astuce** vient du fait que ce procédé ne nécessite pas de connaître explicitement ce nouvel espace : pour résoudre le problème d'optimisation dans le **feature space** on a juste besoin de connaître le **noyau** associé au feature space. D'un point de vu formel un noyau est une fonction

$$K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$$

dont les propriétés sont proches d'un produit scalaire. Il existe donc tout un tas de noyau avec lesquels on peut faire des SVM, par exemple

- **Linéaire** (sur $\mathbb{R}^d$) : $K(x, x') = x^t x'$.
- **Polynomial** (sur $\mathbb{R}^d$) : $K(x, x') = (x^t x' + 1)^d$.
- **Gaussien** (Gaussian radial basis function ou RBF) (sur $\mathbb{R}^d$)

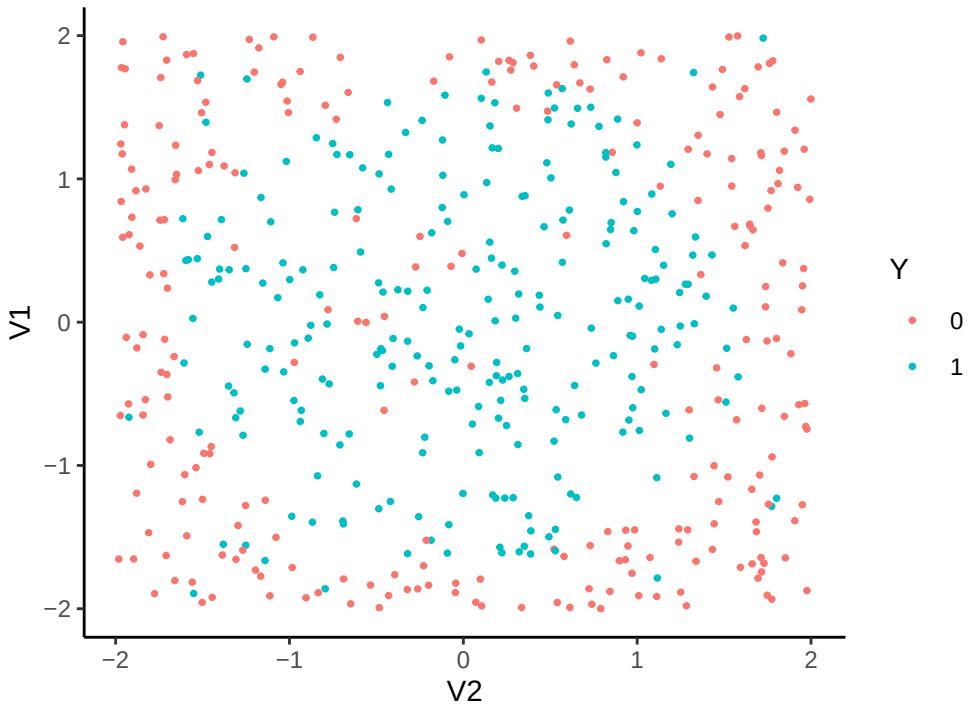
$$K(x, x') = \exp\left(-\frac{\|x - x'\|}{2\sigma^2}\right).$$

- **Laplace** (sur $\mathbb{R}$) : $K(x, x') = \exp(-\gamma|x - x'|)$.
- **Noyau min** (sur $\mathbb{R}^+$) : $K(x, x') = \min(x, x')$.
- ...

Bien entendu, en pratique tout le problème va consister à **trouver le bon noyau** !

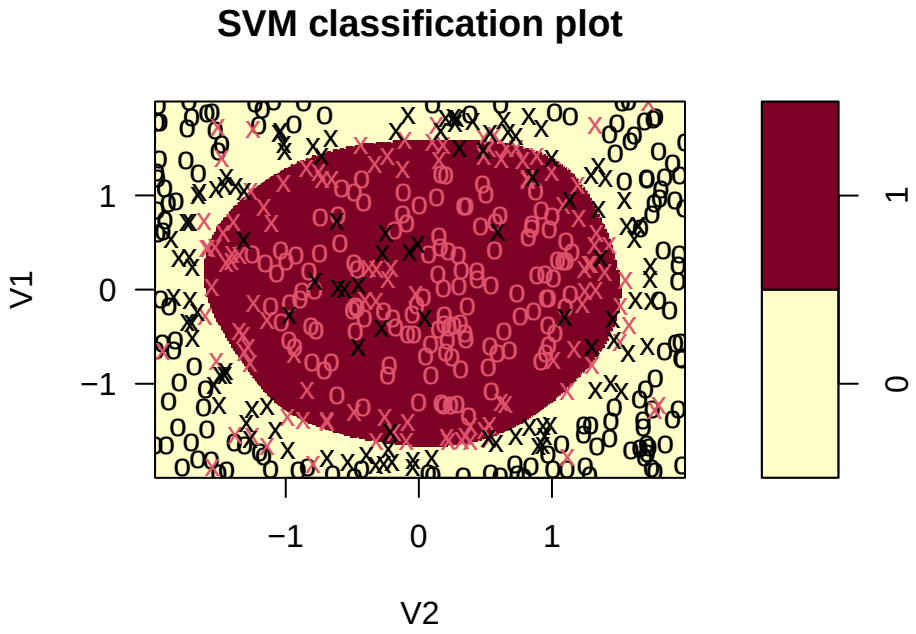
Exercice 5.3 (Non linéarité avec un noyau)**.** On considère le jeu de données suivant où le problème est d'expliquer Y par $V1$ et $V2$.

```
n <- 500
set.seed(13)
X <- matrix(runif(n*2,-2,2),ncol=2) |> as.data.frame()
Y <- rep(0,n)
cond <- (X$V1^2+X$V2^2)<=2.8
Y[cond] <- rbinom(sum(cond),1,0.9)
Y[!cond] <- rbinom(sum(!cond),1,0.1)
df <- X |> mutate(Y=as.factor(Y))
ggplot(df)+aes(x=V2,y=V1,color=Y)+geom_point()+theme_classic()
```



1. Ajuster une svm linéaire et visualiser l'hyperplan séparateur. Que remarquez-vous ?
2. Exécuter la commande suivante et commenter la sortie.

```
mod.svm1 <- svm(Y~.,data=df,kernel="radial",gamma=1,cost=1)
plot(mod.svm1,df,grid=250)
```



3. On fixe le paramètre **gamma** à 10 et on fait varier le paramètre `cost` dans `c(10^(-5),1,10^6)`. Construire les 3 **SVM**.

4. Évaluer les erreurs d'ajustement (en calculant l'accuracy sur les données d'entraînement par exemple) et visualiser les 3 **svm**. Conclure (on pourra également comparer les nombres de vecteur support).

5. Sélectionner automatiquement ces paramètres. On pourra utiliser la fonction **tune** en faisant varier `cost` dans `c(0.1,1,10,100,1000)` et `gamma` dans `c(0.1,0.5,1,10,100)`.

```
tune.out <- tune(svm,Y~.,data=...,kernel="...",
                 ranges=list(cost=...,gamma=...))
```

6. Faire de même en utilisant **tidymodels**. On pourra utiliser l'algorithme suivant :

```
library(tidymodels)
tune_spec <-
  svm_rbf(cost=tune(),rbf_sigma=tune()) |>
  set_mode("classification") |>
  set_engine("kernlab")
```

7. Visualiser la règle sélectionnée. On pourra utiliser `extract_fit_engine`.

5.4 Support vector regression

Dans un contexte de régression (lorsque $y_i \in \mathbb{R}$), on ne recherche plus la l'hyperplan qui va séparer au mieux. On va dans ce cas là cherche à approcher au mieux les valeurs de y_i. Cela revient à chercher $w \in \mathbb{R}^p$ et $b \in \mathbb{R}$ tels que

$$|\langle w, x_i \rangle + b - y_i| \leq \varepsilon$$

avec $\varepsilon > 0$ petit à choisir par l'utilisateur. Par analogie avec la **SVM** binaire, on va ainsi chercher (w, b) qui minimisent

$$\frac{1}{2}\|w\|^2$$
$$\text{sous les contraintes } |y_i - \langle w, x_i \rangle - b| \leq \varepsilon,\ i = 1, \ldots, n,$$

Les contraintes impliquent que toute les observations doivent se définir dans une **marge** ou **bande** de taille 2ε. Cette hypothèse peut amener l'utilisateur à utiliser des valeurs de ε très grandes et empêcher la solution de bien ajuster le nuage de points. Pour pallier à cela, on introduit, comme dans le cas de la SVM binaire, des **variables ressorts** qui vont autoriser certaines observations à se situer en dehors de la marge. Le problème revient alors à trouver $(w, b, \xi, \xi^\star)$ qui minimise

$$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^\star)$$
$$\text{sous les contraintes } \begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i,\ i = 1, \ldots, n, \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^\star,\ i = 1, \ldots, n \\ \xi_i \geq 0, \xi_i^\star \geq 0,\ i = 1, \ldots, n \end{cases}$$

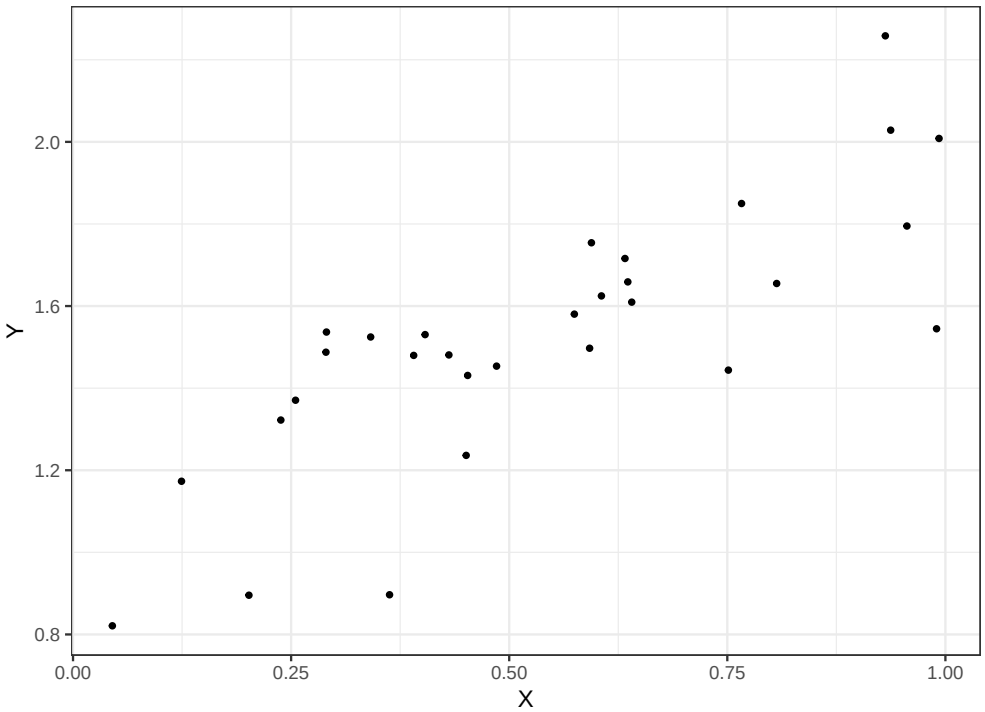
Les solutions s'obtiennent exactement de la même façon que dans le cas binaire. On montre notamment que $w^\star$ s'écrit comme une combinaison linéaire de vecteurs supports :

$$w^\star = \sum_{i=1}^{n}(\alpha_i^\star - \alpha_i)x_i.$$

Les vecteurs supports sont les observations vérifiant $\alpha_i^\star - \alpha_i \neq 0$. Ici encore il faudra calibrer le paramètre C et on pourra utiliser l'astuce du noyau.

Exercice 5.4. On considère le nuage de points $(x_i, y_i), i = 1, \ldots, n$ définie ci-dessous:

```r
set.seed(321)
n <- 30
X <- runif(n)
eps <- rnorm(n,0,0.2)
Y <- 1+X+eps
df <- data.frame(X,Y)
p1 <- ggplot(df)+aes(x=X,y=Y)+geom_point()
p1
```



On souhaite faire une **SVR** permettant de prédire Y par X. On peut l'obtenir sur **R** toujours avec la fonction **svm** de **e1071**:

```r
svr1 <- svm(Y~.,data=df,kernel="linear",epsilon=0.5,cost=100,scale=FALSE)
```

On choisit ici exceptionnellement de ne pas réduire les X.

1. Écrire une fonction **R** qui, à partir d'un objet `svm`, calcule l'équation de la droite de la SVR. Cette fonction pourra également tracer cette doite ainsi que la marge.
2. Comparer la **SVR** précédente avec celle utilisant `epsilon=0.6`.
3. On ajoute le point de coordonnées $(0.05, 3)$ aux données. Discuter de la **SVR** pour ce nouveau jeu de données en utilisant plusieurs valeurs pour `C` et `epsilon`.

   ```r
   df1 <- df |> bind_rows(data.frame(X=0.05,Y=3))
   ```

5.5 SVM sur les données spam

On considère le jeu de données `spam` où le problème est d'expliquer la variable `type` par les autres.

```
library(kernlab)
data(spam)
summary(spam$type)
## nonspam    spam
##    2788    1813
```

On veut comparer plusieurs **svm** en utilisant le package `kernlab`. On pourra trouver un descriptif du package à cette adresse https://www.jstatsoft.org/article/view/v011i09.

1. Utiliser la fonction **ksvm** pour faire une svm linéaire et une svm à noyau gaussien. On prendra comme paramètre 1 pour `C` et pour le paramètre du noyau gaussien.
2. Évaluer la performance des 2 svm précédentes en calculant l'erreur de classification par validation croisée 5 blocs. Comparer ces deux algorithmes.
3. Refaire la svm à noyau gaussien avec l'option `kpar='automatic'`. Expliquer.
4. On s'intéresse maintenant à l'AUC. À partir de validation croisée, sélectionner un noyau (linéaire ou gaussien) ainsi que des valeurs de paramètres associés au noyau, sans oublier le paramètre C. On pourra utiliser le package **tidymodels** et comparer le résultat obtenu à celui d'une forêt aléatoire.

5.6 Exercices

Exercice 5.5 (Résolution du problème d'optimisation dans le cas séparable)**.** On considère n observations $(x_1, y_1), \ldots, (x_n, y_n)$ telles que $(x_i, y_i) \in \mathbb{R}^p \times \{-1, 1\}$. On cherche à expliquer la variable Y par X. On considère l'algorithme SVM et on se place dans le cas où les données sont séparables.

1. Soit $\mathcal{H}$ un hyperplan séparateur d'équation $\langle w, x\rangle + b = 0$ où $w \in \mathbb{R}^p, b \in \mathbb{R}$. Exprimer la distance entre $x_i, i = 1, \ldots, n$ et $\mathcal{H}$ en fonction de w et b.
2. Expliquer la logique du problème d'optimisation

$$\max_{w,b,\|w\|=1} M$$

$$\text{sous les contraintes } y_i(\langle w, x_i\rangle + b) \geq M,\ i = 1, \ldots, n.$$

3. Montrer que ce problème peut se réécrire

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{sous les contraintes } y_i(\langle w, x_i\rangle + b) \geq 1,\ i = 1, \ldots, n.$$

4. On rappelle que pour la minimisation d'une fonction $h : \mathbb{R}^p \to \mathbb{R}$ sous contraintes affines $g_i(u) \geq 0, i = 1, \ldots, n$, le Lagrangien s'écrit

$$L(u, \alpha) = h(u) - \sum_{i=1}^{n} \alpha_i g_i(u).$$

Si on désigne par $u_\alpha = \text{argmin}_u\, L(u, \alpha)$, la fonction duale est alors donnée par

$$\theta(\alpha) = L(u_\alpha, \alpha) = \min_{u \in \mathbb{R}^p} L(u, \alpha),$$

et le problème dual consiste à maximiser $\theta(\alpha)$ sous les contraintes $\alpha_i \geq 0$. En désignant par $\alpha^\star$ la solution de ce problème, on déduit la solution du problème primal $u^\star = u_{\alpha^\star}$. Les conditions de Karush-Kuhn-Tucker sont données par

- $\alpha_i^\star \geq 0$.
- $g_i(u_{\alpha^\star}) \geq 0$.
- $\alpha_i^\star g_i(u_{\alpha^\star}) = 0$.

a. Écrire le Lagrangien du problème considéré et en déduire une expression de w en fonction des α_i et des observations.

b. Écrire la fonction duale.

c. Écrire les conditions KKT et en déduire les solutions $w^\star$ et $b^\star$.

d. Interpréter les conditions KKT.

Exercice 5.6 (Règle svm à partir de sorties R)**.** On considère n observations $(x_1, y_1), \ldots, (x_n, y_n)$ telles que $(x_i, y_i) \in \mathbb{R}^3 \times \{-1, 1\}$. On cherche à expliquer la variable Y par $X = (X_1, X_2, X_3)$. On considère l'algorithme SVM et on se place dans le cas où les données sont séparables. On rappelle que cet algorithme consiste à chercher une droite d'équation $w^t x + b = 0$ où $(w, b) \in \mathbb{R}^3 \times \mathbb{R}$ sont solutions du problème d'optimisation (problème primal)

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

$$\text{sous les contraintes } y_i(w^t x_i + b) \geq 1,\ \ i = 1, \ldots, n.$$

On désigne par $\alpha_i^\star, i = 1, \ldots, n$, les solutions du problème dual et par $(w^\star, b^\star)$ les solutions du problème ci-dessus.

1. Donner la formule permettant de calculer $w^\star$ en fonction des $\alpha_i^\star$.
2. Expliquer comment on classe un nouveau point $x \in \mathbb{R}^3$ par la méthode **svm**.
3. Les données se trouvent dans un dataframe `df`. On exécute

```
set.seed(1234)
n <- 100
X <- data.frame(X1=runif(n),X2=runif(n),X3=runif(n))
X <- data.frame(X1=scale(runif(n)),X2=scale(runif(n)),X3=scale(runif(n)))
Y <- rep(-1,100)
Y[X[,1]<X[,2]] <- 1
#Y <- (apply(X,1,sum)<=0) |> as.numeric() |> as.factor()
df <- data.frame(X,Y=as.factor(Y))

mod.svm <- svm(Y~.,data=df,kernel="linear",cost=10000000000)
```

   et on obtient

```
df[mod.svm$index,]
##       X1   X2   X3  Y
## 51 -1.1 -1.0 -1.0  1
## 92  0.7  0.8  1.1  1
## 31  0.7  0.5 -1.0 -1
## 37 -0.5 -0.6  0.3 -1
mod.svm$coefs
##      [,1]
## [1,]   59
## [2,]   49
## [3,]  -30
## [4,]  -79
mod.svm$rho
## [1] -0.5
```

   Calculer les valeurs de $w^\star$ et $b^\star$. En déduire la règle de classification.

4. On dispose d'une nouvelle observation $x = (1, -0.5, -1)$. Dans quel groupe (`-1` ou `1`) l'algorithme affecte cette nouvelle donnée ?

partie III

Arbres et agrégation d'arbres

6 Méthodes CART

Les méthodes par arbres sont des algorithmes où la prévision s'effectue à partir de **moyennes locales**. Plus précisément, étant donné un échantillon $(x_1, y_1) \ldots, (x_n, y_n)$, l'approche consiste à :

- construire une partition de l'espace de variables explicatives ($\mathbb{R}^p$) ;
- prédire la sortie d'une nouvelle observation x en faisant :
  - la moyenne des y_i pour les x_i qui sont dans la même classe que x si on est en régression ;
  - un vote à la majorité parmi les y_i tels que les x_i qui sont dans la même classe que x si on est en classification.

Bien entendu toute la difficulté est de trouver la "bonne partition" pour le problème d'intérêt. Il existe un grand nombre d'algorithmes qui permettent de trouver une partition. Le plus connu est l'algorithme **CART** (Breiman et al. 1984) où la partition est construite par **divisions successives** au moyen d'hyperplan orthogonaux aux axes de $\mathbb{R}^p$. L'algorithme est récursif : il va à chaque étape séparer un groupe d'observations (**nœuds**) en deux groupes (**nœuds fils**) en cherchant la meilleure variable et le meilleur seuil de coupure. Ce choix s'effectue à partir d'un critère **d'impureté** : la meilleure coupure est celle pour laquelle l'impureté des 2 nœuds fils sera minimale. Nous étudions cet algorithme dans cette partie.

6.1 Coupures CART en fonction de la nature des variables

Une partition CART s'obtient en séparant les observations en 2 selon une coupure parallèle aux axes puis en itérant ce procédé de séparation binaire sur les deux groupes... Par conséquent la première question à se poser est : pour un ensemble de données $(x_1, y_1), \ldots, (x_n, y_n)$ fixé, comment obtenir la meilleure coupure ?

Comme souvent ce sont les données qui vont répondre à cette question. La sélection de la meilleur coupure s'effectue en introduisant une **fonction d'impureté** $\mathcal{I}$ qui va mesurer le degrés d'hétérogénéité d'un nœud $\mathcal{N}$. Cette fonction prendra de

- grandes valeurs pour les nœuds hétérogènes (les valeurs de Y diffèrent à l'intérieur du nœud) ;

- faibles valeurs pour les nœuds homogènes (les valeurs de Y sont proches à l'intérieur du nœud).

On utilise souvent comme fonction d'impureté :

- la **variance** en régression

$$\mathcal{I}(\mathcal{N}) = \frac{1}{|\mathcal{N}|} \sum_{i:x_i \in \mathcal{N}} (y_i - \overline{y}_{\mathcal{N}})^2,$$

  où $\overline{y}_{\mathcal{N}}$ désigne la moyenne des y_i dans $\mathcal{N}$.

- l'impureté de **Gini** en classification binaire

$$\mathcal{I}(\mathcal{N}) = 2p(\mathcal{N})(1 - p(\mathcal{N}))$$

  où $p(\mathcal{N})$ représente la proportion de 1 dans $\mathcal{N}$.

Les coupures considérées par l'algorithme CART sont des hyperplans orthogonaux aux axes de $\mathbb{R}^p$, choisir une coupure revient donc à choisir une variable j parmi les p variables explicatives et un seuil s dans $\mathbb{R}$. On peut donc représenter une coupure par un couple (j, s). Une fois l'impureté définie, on choisira la coupure (j, s) qui **maximise le gain d'impureté** entre le noeud père et ses deux noeuds fils :

$$\Delta(\mathcal{I}) = \mathbf{P}(\mathcal{N})\mathcal{I}(\mathcal{N}) - (\mathbf{P}(\mathcal{N}_1(j,s))\mathcal{I}(\mathcal{N}_1(j,s)) + \mathbf{P}(\mathcal{N}_2(j,s))\mathcal{I}(\mathcal{N}_2(j,s))$$

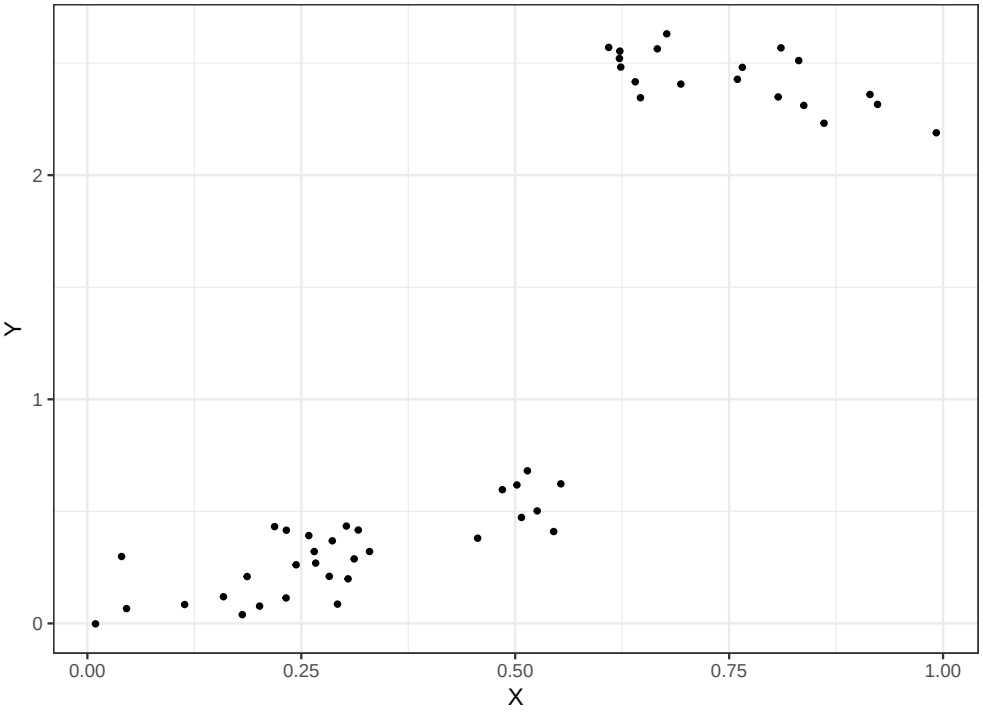
où

- $\mathcal{N}_1(j, s)$ et $\mathcal{N}_2(j, s)$ sont les 2 nœuds fils de $\mathcal{N}$ engendrés par la coupure (j, s) ;
- $\mathbf{P}(\mathcal{N})$ représente la proportion d'observations dans le nœud $\mathcal{N}$.

6.1.1 Arbres de régression

On considère le jeu de données suivant où le problème est d'expliquer la variable quantitative Y par la variable quantitative X.

```
n <- 50
set.seed(1234)
X <- runif(n)
set.seed(5678)
Y <- 1*X*(X<=0.6)+(-1*X+3.2)*(X>0.6)+rnorm(n,sd=0.1)
data1 <- data.frame(X,Y)
ggplot(data1)+aes(x=X,y=Y)+geom_point()
```



1. A l'aide de la fonction **rpart** du package **rpart**, construire un arbre permettant d'expliquer Y par X.

   ```
   library(rpart)
   ```

2. Visualiser l'arbre à l'aide des fonctions **prp** et **rpart.plot** du package **rpart.plot**.

   ```
   library(rpart.plot)
   ```

3. Écrire l'estimateur associé à l'arbre.

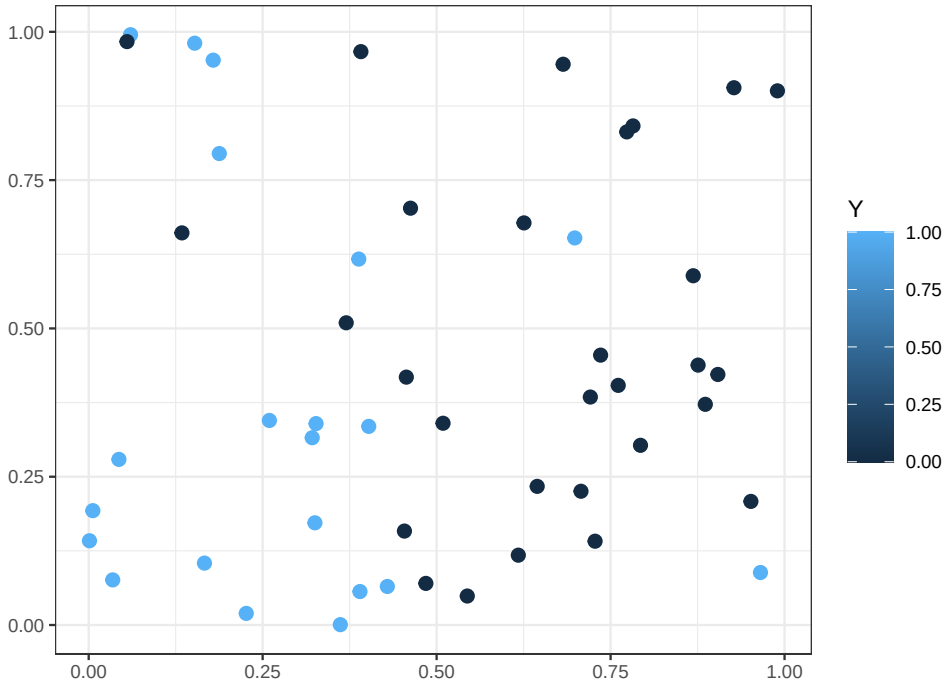
4. Ajouter sur le graphe de la question 1 la partition définie par l'arbre ainsi que les valeurs prédites.

6.1.2 Arbres de classification

On considère les données suivantes où le problème est d'expliquer la variable binaire Y par deux variables quantitatives X_1 et X_2.

```
n <- 50
set.seed(12345)
X1 <- runif(n)
set.seed(5678)
X2 <- runif(n)
```

```
Y <- rep(0,n)
set.seed(54321)
Y[X1<=0.45] <- rbinom(sum(X1<=0.45),1,0.85)
set.seed(52432)
Y[X1>0.45] <- rbinom(sum(X1>0.45),1,0.15)
data2 <- data.frame(X1,X2,Y)
ggplot(data2)+aes(x=X1,y=X2,color=Y)+geom_point(size=2)+
  scale_x_continuous(name="")+
  scale_y_continuous(name="")
```



1. Construire un arbre permettant d'expliquer Y par X_1 et X_2. Représenter l'arbre et identifier l'éventuel problème.
2. Écrire la règle de classification ainsi que la fonction de score définies par l'arbre.
3. Ajouter sur le graphe de la question 1 la partition définie par l'arbre.

6.1.3 Entrée qualitative

On considère les données

```
n <- 100
X <- factor(rep(c("A","B","C","D"),n))
set.seed(1234)
Y[X=="A"] <- rbinom(sum(X=="A"),1,0.9)
```

```
Y[X=="B"] <- rbinom(sum(X=="B"),1,0.25)
Y[X=="C"] <- rbinom(sum(X=="C"),1,0.8)
Y[X=="D"] <- rbinom(sum(X=="D"),1,0.2)
Y <- as.factor(Y)
data3 <- data.frame(X,Y)
```

1. Construire un arbre permettant d'expliquer Y par X.
2. Expliquer la manière dont l'arbre est construit dans ce cadre là.

6.2 Élagage

Le procédé de coupe présenté précédemment permet de définir un très grand nombre d'arbres à partir d'un jeu de données (arbre sans coupure, avec une coupure, deux coupures...). Se pose alors la question de trouver le **meilleur arbre** parmi tous les arbres possibles. Une première idée serait de choisir parmi tous les arbres possibles celui qui optimise un critère de performance. Cette approche, bien que cohérente, n'est généralement pas possible à mettre en œuvre en pratique car le nombre d'arbres à considérer est souvent trop important.

La méthode CART propose une procédure permettant de choisir automatiquement un arbre en 3 étapes :

- On construit un **arbre maximal** (très profond) $\mathcal{T}_{max}$;
- On sélectionne une **suite d'arbres emboités** :

$$\mathcal{T}_{max} = \mathcal{T}_0 \supset \mathcal{T}_1 \supset \cdots \supset \mathcal{T}_K.$$

  La sélection s'effectue en optimisant un critère **Cout/complexité** qui permet de réguler le compromis entre **ajustement** et **complexité** de l'arbre.
- On **sélectionne un arbre** dans cette sous-suite en optimisant un critère de performance.

Cette approche revient à choisir un sous-arbre de l'arbre $\mathcal{T}_{\max}$, c'est-à-dire à enlever des branches à $T_{\max}$, c'est pourquoi on parle **d'élagage**.

6.2.1 Élagage pour un problème de régression

On considère les données **Carseats** du package **ISLR**.

```
library(ISLR)
data(Carseats)
summary(Carseats)
##      Sales          CompPrice       Income
##  Min.   : 0.000   Min.   : 77   Min.   : 21.00
##  1st Qu.: 5.390   1st Qu.:115   1st Qu.: 42.75
##  Median : 7.490   Median :125   Median : 69.00
```

```
##  Mean   : 7.496   Mean   :125   Mean   : 68.66
##  3rd Qu.: 9.320   3rd Qu.:135   3rd Qu.: 91.00
##  Max.   :16.270   Max.   :175   Max.   :120.00
##   Advertising       Population        Price
##  Min.   : 0.000   Min.   : 10.0   Min.   : 24.0
##  1st Qu.: 0.000   1st Qu.:139.0   1st Qu.:100.0
##  Median : 5.000   Median :272.0   Median :117.0
##  Mean   : 6.635   Mean   :264.8   Mean   :115.8
##  3rd Qu.:12.000   3rd Qu.:398.5   3rd Qu.:131.0
##  Max.   :29.000   Max.   :509.0   Max.   :191.0
##   ShelveLoc        Age          Education    Urban
##  Bad    : 96   Min.   :25.00   Min.   :10.0   No :118
##  Good   : 85   1st Qu.:39.75   1st Qu.:12.0   Yes:282
##  Medium:219   Median :54.50   Median :14.0
##               Mean   :53.32   Mean   :13.9
##               3rd Qu.:66.00   3rd Qu.:16.0
##               Max.   :80.00   Max.   :18.0
##    US
##  No :142
##  Yes:258
##
##
##
##
```

On cherche ici à expliquer la variable quantitative **Sales** par les autres variables.

1. Construire un arbre permettant de répondre au problème.
2. Expliquer les sorties de la fonction **printcp** appliquée à l'arbre de la question précédente et calculer le dernier terme de la colonne **rel error**.
3. Construire une suite d'arbres plus grandes en jouant sur les paramètres `cp` et `minsplit` de la fonction **rpart**.
4. Expliquer la sortie de la fonction **plotcp** appliquée à l'arbre de la question précédente.
5. Sélectionner le "meilleur" arbre dans la suite construite.
6. Visualiser l'arbre choisi (utiliser la fonction **prune**).
7. On souhaite prédire les valeurs de Y pour de nouveaux individus à partir de l'arbre sélectionné. Pour simplifier on considèrera ces 4 individus :

```
(new_ind <- Carseats |> as_tibble() |>
   slice(3,58,185,218) |> dplyr::select(-Sales))
## # A tibble: 4 x 10
##   CompPrice Income Advertising Population Price ShelveLoc
##       <dbl>  <dbl>       <dbl>      <dbl> <dbl> <fct>
## 1       113     35          10        269    80 Medium
```

```
## 2        93     91           0         22   117 Bad
## 3       132     33           7         35    97 Medium
## 4       106     44           0        481   111 Medium
## # i 4 more variables: Age <dbl>, Education <dbl>,
## #   Urban <fct>, US <fct>
```

Calculer les valeurs prédites.

8. Séparer les données en un échantillon d'apprentissage de taille 250 et un échantillon test de taille 150.
9. On considère la suite d'arbres définie par

```
set.seed(4321)
tree <- rpart(Sales~.,data=train,cp=0.000001,minsplit=2)
```

Dans cette suite, sélectionner

- un arbre très simple (avec 2 ou 3 coupures)
- un arbre très grand
- l'arbre optimal (avec la procédure d'élagage classique).

10. Calculer l'erreur quadratique de ces 3 arbres en utilisant l'échantillon test.
11. Refaire la comparaison avec une validation croisée 10 blocs.

6.2.2 Élagage en classification binaire et matrice de coût

On considère ici les mêmes données que précédemment mais on cherche à expliquer une version binaire de la variable **Sales**. Cette nouvelle variable, appelée `High` prend pour valeurs `No` si `Sales` est inférieur ou égal à 8, `Yes` sinon. On travaillera donc avec le jeu `data1` défini ci-dessous.

```
High <- ifelse(Carseats$Sales<=8,"No","Yes")
data1 <- Carseats |> dplyr::select(-Sales) |> mutate(High)
```

1. Construire un arbre permettant d'expliquer `High` par les autres variables (sans `Sales` évidemment !) et expliquer les principales différences par rapport à la partie précédente précédente.
2. Expliquer l'option `parms` dans la commande :

```
tree1 <- rpart(High~.,data=data1,parms=list(split="information"))
tree1$parms
## $prior
##    1    2
## 0.59 0.41
##
```

```
## $loss
##      [,1] [,2]
## [1,]    0    1
## [2,]    1    0
##
## $split
## [1] 2
```

3. Expliquer les sorties de la fonction **printcp** sur le premier arbre construit et retrouver la valeur du dernier terme de la colonne **rel error**.

4. Sélectionner un arbre optimal dans la suite.

5. On considère la suite d'arbres

```
tree2 <- rpart(High~.,data=data1,
               parms=list(loss=matrix(c(0,5,1,0),ncol=2)),
               cp=0.01,minsplit=2)
```

Expliquer les sorties des commandes suivantes. On pourra notamment calculer le dernier terme de la colonne **rel error** de la table **cptable**.

```
tree2$parms
## $prior
##    1    2
## 0.59 0.41
##
## $loss
##      [,1] [,2]
## [1,]    0    1
## [2,]    5    0
##
## $split
## [1] 1
printcp(tree2)
##
## Classification tree:
## rpart(formula = High ~ ., data = data1, parms = list(loss = matrix(c(0,
##     5, 1, 0), ncol = 2)), cp = 0.01, minsplit = 2)
##
## Variables actually used in tree construction:
## [1] Advertising Age         CompPrice   Education
## [5] Income      Population  Price       ShelveLoc
##
## Root node error: 236/400 = 0.59
##
## n= 400
##
##          CP nsplit rel error xerror    xstd
```

```
## 1  0.101695      0   1.00000 5.0000 0.20840
## 2  0.050847      2   0.79661 3.8136 0.20909
## 3  0.036017      3   0.74576 3.2034 0.20176
## 4  0.035311      5   0.67373 3.1271 0.20038
## 5  0.025424      9   0.50847 2.6144 0.19069
## 6  0.016949     11   0.45763 2.3475 0.18307
## 7  0.015537     16   0.37288 2.1992 0.17905
## 8  0.014831     21   0.28814 2.1992 0.17905
## 9  0.010593     23   0.25847 2.0466 0.17367
## 10 0.010000     25   0.23729 2.0297 0.17292
```

6. Comparer les valeurs ajustées par les deux arbres considérés.

6.2.3 Calcul de la sous-suite d'arbres optimaux

Exercice 6.1 (Minimisation du critère coût/complexité)**.** On considère l'algorithme qui permet de calculer les suites $(\alpha_m)_m$ et $(T_{\alpha_m})_m$ du théorème présenté en cours. Pour simplifier on se place en classification binaire et on considère les notations suivantes (en plus de celles présentées dans le chapitre) :

- $R(t)$: erreur de classification dans le nœud t pondérée par la proportion d'individus dans le nœud (nombre d'individus dans t sue le nombre total d'individus).
- T^t : la branche de l'arbre T issue du nœud interne t.
- $R(T^t)$: l'erreur de la branche T^t pondérée par la proportion d'individus dans le nœud.

L'algorithme suivant présente le calcul explicite des suites $(\alpha_m)_m$ et $(T_{\alpha_m})_m$.

Initialisation : on pose $\alpha_0 = 0$ et on calcule l'arbre maximale T_0 qui minimise $C_0(T)$. On fixe $m = 0$. Répéter jusqu'à obtenir l'arbre racine

1. Calculer pour tous les nœuds t internes de T_{α_m}
$$g(t) = \frac{R(t) - R(T^t_{\alpha_m})}{|T^t_{\alpha_m}| - 1}$$
2. Choisir le nœud interne t_m qui minimise $g(t)$.
3. On pose
$$\alpha_{m+1} = g(t_m) \quad \text{et} \quad T_{\alpha_{m+1}} = T_{\alpha_m} - T^{t_m}_{\alpha_m}.$$
4. Mise à jour : $m := m + 1$.

Retourner : les suites finies $(\alpha_m)_m$ et $(T_{\alpha_m})_m$.

On propose d'utiliser cet algorithme sur l'arbre construit suivant

```r
gen_class_bin2D <- function(n=100,graine=1234,bayes=0.1){
  set.seed(graine)
  grille <- 0.1
  X1 <- runif(n)
  X2 <- runif(n)
  Y <- rep(0,n)
  cond0 <- (X1>0.2 & X2>=0.8) | (X1>0.6 & X2<0.4) | (X1<0.25 & X2<0.5)
  cond1 <- !cond0
  Y[cond0] <- rbinom(sum(cond0),1,bayes)
  Y[cond1] <- rbinom(sum(cond1),1,1-bayes)
  donnees <- tibble(X1,X2,Y=as.factor(Y))
  px1 <- seq(0,1,by=grille)
  px2 <- seq(0,1,by=grille)
  px <- expand.grid(X1=px1,X2=px2)
  py <- rep(0,nrow(px))
  cond0 <- (px[,1]>0.2 & px[,2]>=0.8) |
    (px[,1]>0.6 & px[,2]<0.4) |
    (px[,1]<0.25 & px[,2]<0.5)
  cond1 <- !cond0
  py[cond0] <- 0
  py[cond1] <- 1
  df <- px |> as_tibble() |> mutate(Y=as.factor(py))
  p <- ggplot(df)+aes(x=X1,y=X2,fill=Y)+
    geom_raster(hjust=1,vjust=1)
  return(list(donnees=donnees,graphe=p))
}
don.2D.arbre <- gen_class_bin2D(n=150,graine=3210,bayes=0.05)$donnees
set.seed(123)
T0 <- rpart(Y~.,data=don.2D.arbre)
rpart.plot(T0,extra = 1)
```

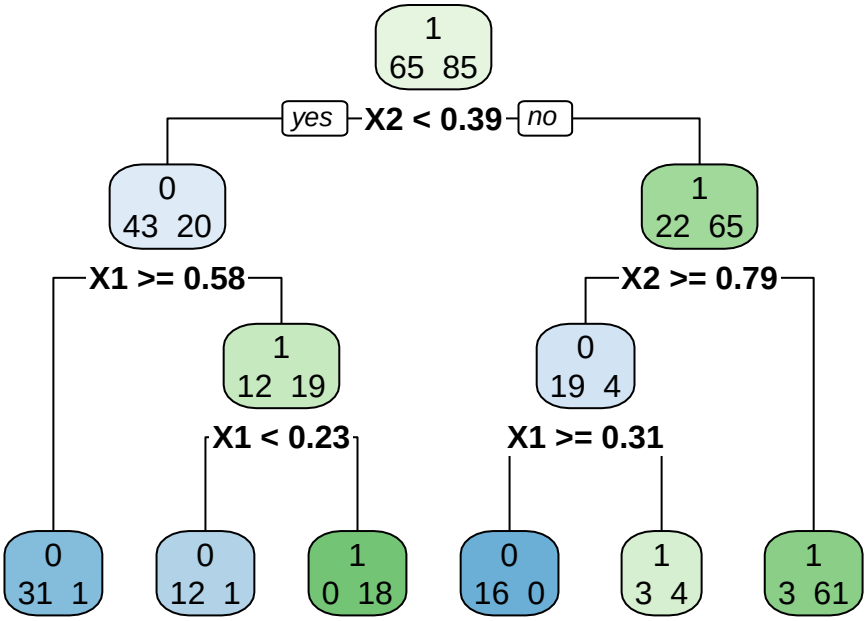


Figure 6.1: L'arbre T_0.

Cet arbre n'est pas l'arbre maximal mais la manière d'élaguer est identique.

1. Calculer pour les 5 nœuds internes de T_0 la fonction $g(t)$.
2. En déduire la valeur de α_1 ainsi que l'arbre T_{α_1}.
3. Retrouver cette valeur en utilisant la fonction `printcp` et représenter l'arbre T_1 en utilisant `prune`.
4. Faire le même travail pour calculer α_2 et T_{α_2}.

7 Forêts aléatoires

Les méthodes par arbres présentées précédemment sont des algorithmes qui possèdent tout un tas de qualités (facile à mettre en œuvre, interprétable...). Ce sont néanmoins rarement les algorithmes qui se révèlent les plus performants. Les méthodes d'agrégation d'arbres présentées dans cette partie sont souvent beaucoup plus pertinentes, notamment en terme de qualité de prédiction. Elles consistent à construire un très grand nombre d'arbres "simples" : $g_1, \ldots, g_B$ et à les agréger en faisant la moyenne :

$$\frac{1}{B}\sum_{k=1}^{B} g_k(x).$$

Les forêts aléatoires (Breiman 2001) et le gradient boosting (Friedman 2001) utilisent ce procédé d'agrégation. Dans ce chapitre on étudiera ces algorithmes sur le je de données `spam` :

```
library(kernlab)
data(spam)
set.seed(1234)
spam <- spam[sample(nrow(spam)),]
```

Le problème est d'expliquer la variable binaire **type** par les autres.

L'algorithme des forêts aléatoires consiste à construire des arbres sur des échantillons bootstrap et à les agréger. Il peut s'écrire de la façon suivante :

Entrées :

- $x \in \mathbb{R}^d$ l'observation à prévoir, $\mathcal{D}_n$ l'échantillon ;
- B nombre d'arbres ; n_{max} nombre max d'observations par nœud
- $m \in \{1, \ldots, d\}$ le nombre de variables candidates pour découper un nœud.

Algorithme : pour $k = 1, \ldots, B$:

1. Tirer un échantillon *bootstrap* dans $\mathcal{D}_n$
2. Construire un *arbre CART sur cet échantillon bootstrap*, chaque coupure est sélectionnée en minimisant la fonction de coût de CART sur un ensemble de m variables choisies au hasard parmi les d. On note $T(., \theta_k, \mathcal{D}_n)$ l'arbre construit.

Sortie : l'estimateur $T_B(x) = \frac{1}{B}\sum_{k=1}^{B} T(x, \theta_k, \mathcal{D}_n)$.

Cet algorithme peut être utilisé sur **R** avec la fonction **randomForest** du package **randomForest** ou la fonction **ranger** du package **ranger**.

Exercice 7.1 (Biais et variance des algorithmes bagging)**.** Comparer le biais et la variance de la forêt $T_B(x)$ au biais et à la variance d'un arbre de la forêt $T(x, \theta_k, \mathcal{D}_n)$. On pourra utiliser $\rho(x) = \text{corr}(T(x, \theta_1, \mathcal{D}_n), T(x, \theta_2, \mathcal{D}_n))$ pour comparer les variances.

Exercice 7.2 (RandomForest versus ranger)**.** On sépare le jeu de données `spam` en un échantillon d'apprentissage et un échantillon test :

```
set.seed(1234)
library(tidymodels)
data_split <- initial_split(spam, prop = 3/4)
spam.app <- training(data_split)
spam.test  <- testing(data_split)
```

1. Entraîner une forêt aléatoire sur les données d'apprentissage uniquement en utilisant les paramètres par défaut de la fonction **randomForest**. Commenter.
2. Calculer les groupes prédits pour les individus de l'échantillon test et en déduire une estimation de l'erreur de classification.
3. Calculer les estimations de la probabilité de spam pour les individus de l'échantillon test.
4. Refaire les questions précédentes avec la fonction **ranger** du package **ranger** (voir https://arxiv.org/pdf/1508.04409.pdf).

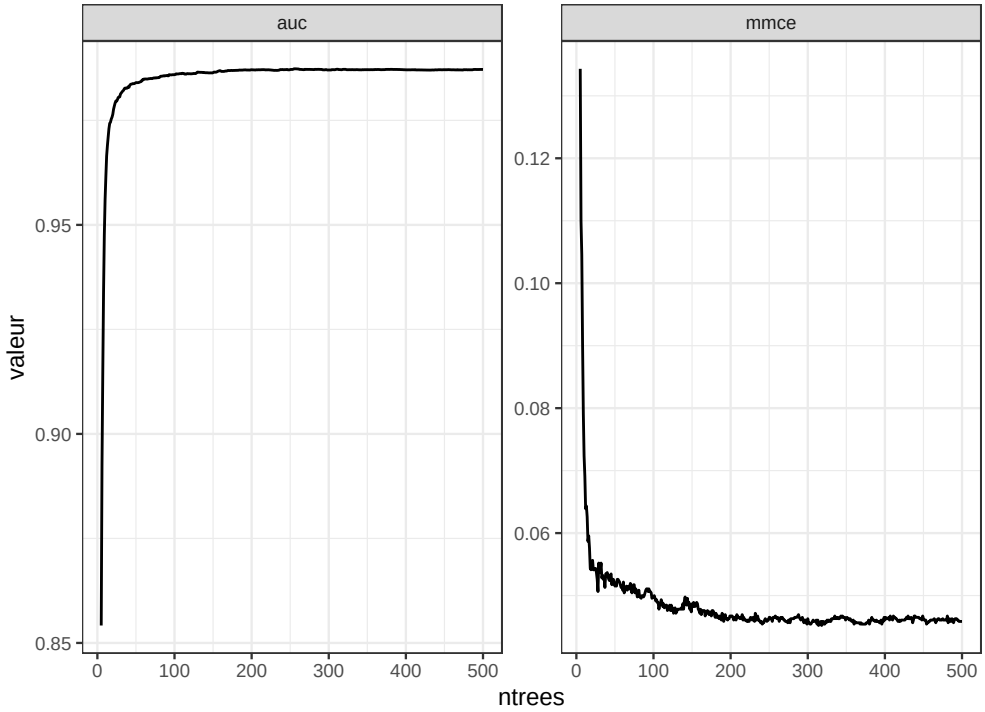
   ```
   library(ranger)
   ```

5. Comparer les temps de calcul de **randomForest** et **ranger**.

Exercice 7.3 (Sélection des paramètres)**.** Nous nous intéressons ici au choix des paramètres de la forêt aléatoire.

1. Expliquer la sortie suivante.

   ```
   set.seed(12345)
   library(OOBCurve)
   foret1 <- ranger(type~.,data=spam,keep.inbag=TRUE)
   spam.task <- mlr::makeClassifTask(data=spam,target="type")
   erreurs <- OOBCurve(foret1,measures = list(mmce, auc),
                   task=spam.task,data=spam)
   erreurs1 <- erreurs |> as_tibble() |> mutate(ntrees=1:500) |>
     filter(ntrees>=5) |>
     pivot_longer(-ntrees,names_to="Erreur",values_to="valeur")
   ggplot(erreurs1)+aes(x=ntrees,y=valeur)+geom_line()+
     facet_wrap(~Erreur,scales="free")
   ```



2. Construire la forêt avec `mtry=1` et comparer ses performances avec celle construite précédemment.
3. A l'aide des outils `tidymodels` sélectionner les paramètres `mtry` et `min_n` dans les grilles `c(1,6,seq(10,50,by=10),57)` et `c(1,5,100,500)`. On pourra notamment visualiser les critères en fonction des valeurs de paramètres.
4. Visualiser l'importance des variables pour les scores d'impureté et de permutations.

Exercice 7.4 (Arbre vs forêt aléatoire)**.** Proposer et mettre en œuvre une procédure permettant de comparer les performances (courbes ROC, AUC et accuracy) d'un arbre CART utilisant la procédure d'élagage proposée dans la section 6.2 avec une forêt aléatoire.

8 Gradient boosting

Les algorithmes de gradient boosting permettent de minimiser des pertes empiriques de la forme

$$\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i)).$$

où $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ est une fonction de coût convexe en son second argument. Il existe plusieurs type d'algorithmes boosting. Un des plus connus et utilisés a été proposé par Friedman (2001), c'est la version que nous étudions dans cette partie.

Cette approche propose de chercher la meilleure combinaison linéaire d'arbres binaires, c'est-à-dire que l'on recherche $g(x) = \sum_{m=1}^{M} \alpha_m h_m(x)$ qui minimise

$$\mathcal{R}_n(g) = \frac{1}{n}\sum_{i=1}^{n}\ell(y_i, g(x_i)).$$

Optimiser sur toutes les combinaisons d'arbres binaires se révélant souvent trop compliqué, Friedman (2001) utilise une descente de gradient pour construire la combinaison d'arbres de façon récursive. L'algorithme est le suivant :

Entrées :

- $d_n = (x_1, y_1), \ldots, (x_n, y_n)$ l'échantillon, λ un paramètre de régularisation tel que $0 < \lambda \leq 1$.
- $M \in \mathbb{N}$ le nombre d'itérations.
- paramètres de l'arbre (nombre de coupures...)

Itérations :

1. Initialisation : $g_0(.) = \text{argmin}_c \frac{1}{n}\sum_{i=1}^{n}\ell(y_i, c)$
2. Pour $m = 1$ à M :
   a. Calculer l'opposé du gradient $-\frac{\partial}{\partial g(x_i)}\ell(y_i, g(x_i))$ et l'évaluer aux points $g_{m-1}(x_i)$:
   $$U_i = -\frac{\partial}{\partial g(x_i)}\ell(y_i, g(x_i))\Big|_{g(x_i)=g_{m-1}(x_i)}, \quad i = 1, \ldots, n.$$
   b. Ajuster un arbre sur l'échantillon $(x_1, U_1), \ldots, (x_n, U_n)$, on le note h_m.
   c. Mise à jour : $g_m(x) = g_{m-1}(x) + \lambda h_m(x)$.

Sortie : la suite $(g_m(x))_m$.

Sur **R** On peut utiliser différents packages pour faire du gradient boosting. Nous utilisons ici le package **gbm** (Ridgeway 2006).

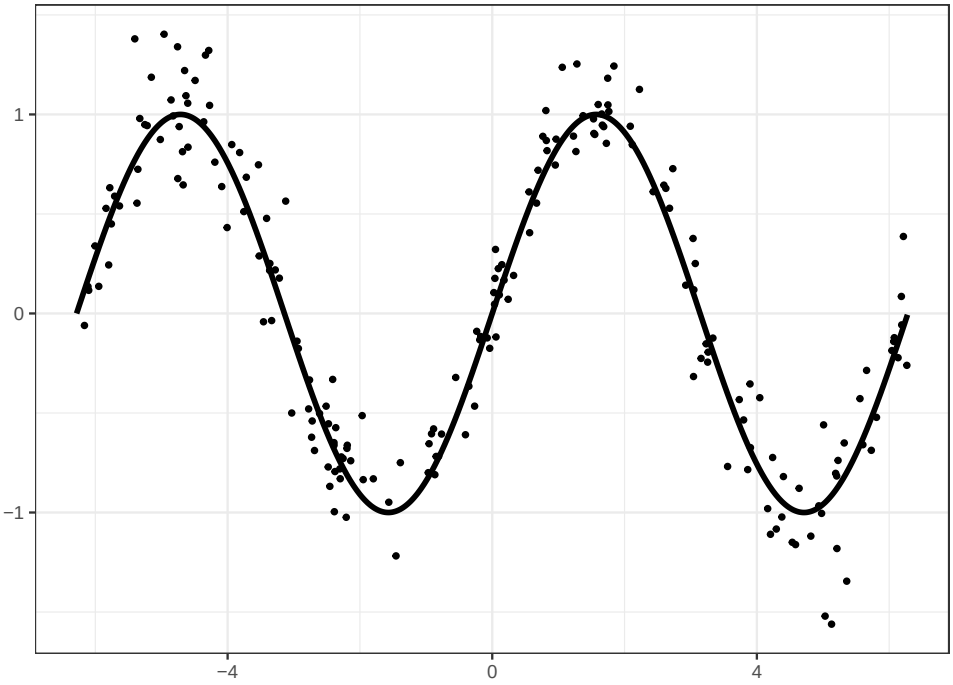
8.1 Un exemple simple en régression

On considère un jeu de données $(x_i, y_i), i = 1, \ldots, 200$ issu d'un modèle de régression

$$y_i = m(x_i) + \varepsilon_i$$

où la vraie fonction de régression est la fonction **sinus** (mais on va faire comme si on ne le savait pas).

```
x <- seq(-2*pi,2*pi,by=0.01)
y <- sin(x)
set.seed(1234)
X <- runif(200,-2*pi,2*pi)
Y <- sin(X)+rnorm(200,sd=0.2)
df1 <- tibble(X,Y)
df2 <- tibble(X=x,Y=y)
p1 <- ggplot(df1)+aes(x=X,y=Y)+geom_point()+
  geom_line(data=df2,linewidth=1)+xlab("")+ylab("")
p1
```



1. Rappeler ce que siginifie le L_2-boosting.

2. A l'aide de la fonction **gbm** du package **gbm** construire un algorithme de L_2-boosting. On utilisera 500000 itérations et gardera les autres valeurs par défaut de paramètres, à l'exception de `bag.fraction` qu'on prendra égal à 1.

```
library(gbm)
L2boost <- gbm(Y~.,data=df1,...)
```

3. Visualiser l'estimateur à la première itération. On pourra faire un **predict** avec l'option `n.trees` ou utiliser directement la fonction **plot.gbm** avec l'option `n.trees`.

```
help(predict.gbm)
prev1 <- predict(L2boost,...)
#ou
help(plot.gbm)
plot(L2boost,...)
```

4. Faire de même pour les itérations 1000 et 500000.

```
prev1000 <- predict(L2boost,newdata=df2,...)
prev500000 <- predict(L2boost,newdata=df2,...)
...
```

5. Sélectionner le nombre d'itérations par la procédure de votre choix.

```
L2boost.sel <- gbm(Y~.,data=df1,...)
gbm.perf(...)
```

6. Représenter l'estimateur sélectionné.

```
prev_opt <- predict(L2boost.sel,newdata=df2,...)
...
```

8.2 Adaboost et logitboost pour la classification binaire.

On considère le jeu de données **spam** du package **kernlab**.

```
library(kernlab)
data(spam)
set.seed(1234)
spam <- spam[sample(nrow(spam)),]
```

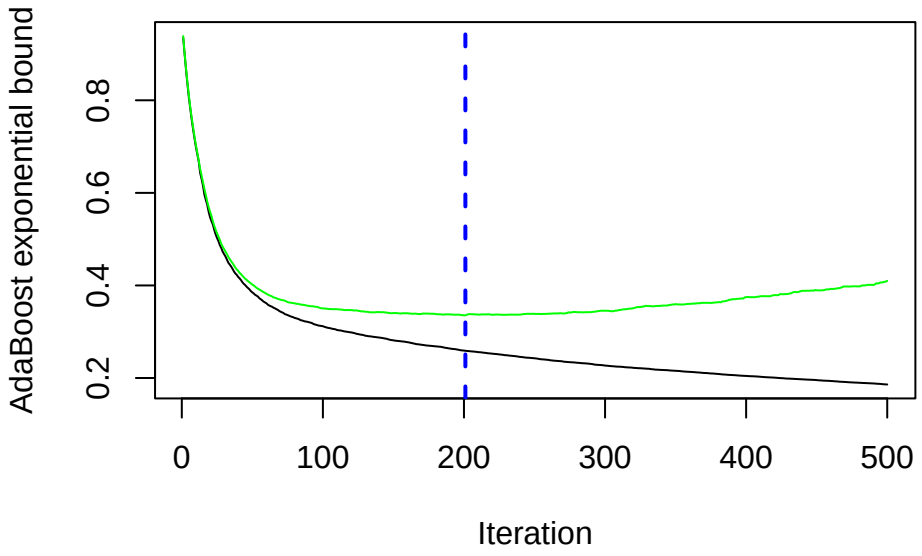
1. Exécuter la commande et commenter la sortie.

```
model_ada1 <- gbm(type~.,data=spam,distribution="adaboost",
                  interaction.depth=2,
                  shrinkage=0.05,n.trees=500)
```

2. Proposer une correction permettant de faire fonctionner l'algorithme.

```
spam1 <- spam
spam1$type <- ...
...
```

3. Expliciter le modèle ajusté par la commande précédente.
4. Effectuer un **summary** du modèle ajusté. Expliquer la sortie.
5. Utiliser la fonction **vip** du package **vip** pour retrouver ce sorties.
6. Sélectionner le nombre d'itérations pour l'algorithme adaboost en faisant une validation croisée 5 blocs.




```
## [1] 201
```

7. Pour l'estimateur sélectionné, calculer la prévision (label et probabilité d'être un spam) de l'individu suivant :

```
xnew <- spam[1000,-58]
```

8. Faire la même procédure en changeant la valeur du paramètre **shrinkage** (par exemple 0.05 et 0.5). Interpréter.
9. Expliquer la différence entre **adaboost** et **logitboost** et précisez comment on peut mettre en œuvre ce dernier algorithme.

8.3 Comparaison de méthodes

On reprend les données `spam` de l'exercice précédent et on les coupe en un échantillon d'apprentissage pour entraîner les algorithmes et un échantillon test pour les comparer :

```
set.seed(1234)
perm <- sample(1:nrow(spam))
app <- spam[perm[1:3000],]
test <- spam[-perm[1:3000],]
```

1. Sur les données d'apprentissage uniquement, entraîner
   - l'algorithme **adaboost** en sélectionnant le nombre d'itérations par validation croisée
   - l'algorithme **logitboost** en sélectionnant le nombre d'itérations par validation croisée
   - une **forêt aléatoire** avec les paramètres par défaut
   - un **arbre CART**

   ```
   app1 <- app |> mutate(type=as.numeric(type)-1)
   model_ada <- gbm(type~.,data=app1,...)
   nb_ada <- gbm.perf(...)

   model_logit <- gbm(type~.,data=app1,...)
   nb_logit <- gbm.perf(...)

   library(ranger)
   foret <- ranger(...,probability=TRUE)
   library(rpart)
   arbre <- rpart(...)
   cp_opt <- ...
   arbre.opt <- prune(...)
   ```

2. Pour les 4 algorithmes, calculer, pour tous les individus de l'échantillon `test`, la probabilité que ce soit un spam. On pourra stocker toutes ces probabilités dans un même `tibble`.

   ```
   prob <- tibble(ada=predict(...,type="response"),
                  logit=predict(...),
                  foret=predict(...)$prediction[,2],
                  arbre=predict(...)[,2],
                  obs=test$type)
   ```

3. Comparer les 3 algorithmes avec la courbe ROC, l'AUC et l'erreur de classification.
4. Comment aurait-on pu faire pour obtenir des résultats plus précis ?

8.4 Xgboost

L'algorithme **xgboost** (Chen et Guestrin 2016) va plus loin que le **gradient boosting** en minimisant une approximation à l'ordre 2 de la fonction de perte et en ajoutant un terme de régularisation dans la fonction objectif. On cherche toujours des combinaisons d'arbres

$$f_b(x) = f_{b-1}(x) + h_b(x) \quad \text{où} \quad h_b(x) = w_{q(x)}$$

est un arbre à T feuilles : $w \in \mathbb{R}^T$ et $q : \mathbb{R}^d \to \{1, 2, \ldots, T\}$. À l'étape b, on cherche l'arbre qui minimise la **fonction objectif** de la forme

$$\begin{aligned} \text{obj}^{(b)} &= \sum_{i=1}^{n} \ell(y_i, f_b(x_i)) + \sum_{j=1}^{b} \Omega(h_j) \\ &= \sum_{i=1}^{n} \ell(y_i, f_{b-1}(x_i) + h_b(x_i)) + \sum_{j=1}^{b} \Omega(h_j) \end{aligned}$$

où $\Omega(h_j)$ est un terme de **régularisation** qui va pénaliser h_j en fonction de son nombre de feuilles T et des valeurs prédites w. Un développement limité à l'ordre 2 permet d'approcher cette fonction par

$$\text{obj}^{(b)} = \sum_{i=1}^{n} [\ell_i^{(1)} h_b(x_i) + \frac{1}{2} \ell_i^{(2)} h_b^2(x_i)] + \Omega(h_b) + \text{constantes},$$

avec

$$\ell_i^{(1)} = \frac{\partial \ell(y_i, f(x))}{\partial f(x)}(f_{b-1}(x_i)) \quad \text{et} \quad \ell_i^{(2)} = \frac{\partial^2 \ell(y_i, f(x))}{\partial f(x)^2}(f_{b-1}(x_i)).$$

Pour les arbres, la fonction de régularisation a la forme suivante :

$$\Omega(h) = \Omega(T, w) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2,$$

où γ et λ contrôlent le **poids** que l'on donne aux paramètres de l'arbre. On obtient au final l'algorithme suivant

1. Initialisation $f_0 = h_0$.
2. Pour $b = 1, \ldots, B$

a) Ajuster un arbre h_b à T feuilles qui minimise

$$\sum_{i=1}^{n}[\ell_i^{(1)}h_b(x_i) + \frac{1}{2}\ell_i^{(2)}h_b^2(x_i)] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j.$$

b) Mettre à jour

$$f_b(x) = f_{b-1}(x) + h_b(x).$$

3. Sortie : la suite d'algorithmes $(f_b)_b$.

On pourra trouver plus de précisions ici : https://xgboost.readthedocs.io/en/stable/tutorials/index.html

Exercice 8.1 (Prise en main des principales fonctions de xgboost)**.** On commence par charger le package

```
library(xgboost)
```

et on reprend les données sur le **sinus** de la section 8.1 :

```
x <- seq(-2*pi,2*pi,by=0.01)
y <- sin(x)
set.seed(1234)
X <- runif(200,-2*pi,2*pi)
Y <- sin(X)+rnorm(200,sd=0.2)
df1 <- data.frame(X,Y)
df2 <- data.frame(X=x,Y=y)
```

La fonction `xgboost` requiert que les données possèdent la classe `xgb.DMatrix`, on peut l'obtenir avec

```
X_mat <- xgb.DMatrix(as.matrix(df1[,1]),label=df1$Y)
```

1. Expliquer la sortie

```
boost1 <- xgboost(data=X_mat,nrounds = 5,
                  params=list(objective="reg:squarederror"))
## [1]  train-rmse:0.633250
## [2]  train-rmse:0.468674
## [3]  train-rmse:0.354599
## [4]  train-rmse:0.275842
## [5]  train-rmse:0.224803
boost1
## ##### xgb.Booster
## raw: 16.5 Kb
```

```
## call:
##   xgb.train(params = params, data = dtrain, nrounds = nrounds,
##     watchlist = watchlist, verbose = verbose, print_every_n = print_every_n,
##     early_stopping_rounds = early_stopping_rounds, maximize = maximize,
##     save_period = save_period, save_name = save_name, xgb_model = xgb_model,
##     callbacks = callbacks)
## params (as set within xgb.train):
##   objective = "reg:squarederror", validate_parameters = "TRUE"
## xgb.attributes:
##   niter
## callbacks:
##   cb.print.evaluation(period = print_every_n)
##   cb.evaluation.log()
## niter: 5
## nfeatures : 1
## evaluation_log:
##   iter train_rmse
##  <num>      <num>
##      1  0.6332497
##      2  0.4686737
##      3  0.3545990
##      4  0.2758424
##      5  0.2248031
```

2. Faire la même chose avec 100 itération et un `learning rate` de 0.1.

3. On peut obtenir les valeurs prédites entre -2π et 2π pour 100 itérations avec

   ```
   Xtest <- as.matrix(df2$X)
   prev100 <- predict(boost2,newdata=Xtest,iterationrange = c(1,101))
   ```

   Tracer les estimateurs **xgboost** pour 1 itération, 20 itérations et 100 itérations. Commenter.

4. Commenter la sortie

   ```
   set.seed(123)
   sel.xgb <- xgb.cv(data = X_mat,
                     nrounds = 100, objective = "reg:squarederror",
                     eval_metric = "rmse",
                     nfold=5,eta=0.1,
                     early_stopping_rounds=10,
                     verbose=FALSE)
   sel.xgb$evaluation_log |> head()
   ##     iter train_rmse_mean train_rmse_std test_rmse_mean
   ##    <num>           <num>          <num>          <num>
   ## 1:     1       0.7894653    0.012973111      0.7906004
   ## 2:     2       0.7189852    0.011889426      0.7228767
   ## 3:     3       0.6556685    0.010877136      0.6607388
   ```

```
## 4:     4       0.5985454    0.010104324      0.6055111
## 5:     5       0.5471973    0.009379501      0.5567722
## 6:     6       0.5007942    0.008971210      0.5129458
##    test_rmse_std
##            <num>
## 1:    0.05513590
## 2:    0.05364182
## 3:    0.05284791
## 4:    0.05240736
## 5:    0.05117564
## 6:    0.04971156
(ite.opt.xgb <- sel.xgb$best_iteration)
## [1] 27
sel.xgb$niter
## [1] 37
```

5. Tracer les prévisions pour l'algorithme sélectionné.

Exercice 8.2 (Xgboost sur les données spam)**.** On reprend les données spam de la section 8.2. Entraîner un algorithme **xgboost** avec la fonction de perte `binary:logistic` et en sélectionnant la nombre d'itérations par validation croisée en optimisant l'AUC. **Attention** cette fonction de perte requiert que la variable à expliquer prenne pour valeurs 0 ou 1 en classe `numeric`.

Exercice 8.3 (Sélection avec tidymodels)**.** Refaire l'exercice précédent avec la syntaxe `tidymodels`. On choisira notamment :

- la profondeur des arbres dans le vecteur

  ```
  c(1,3,8,10)
  ```

- le nombre d'itérations entre 1 et 500 avec un `early_stopping` toujours égal à 10 et un learning rate à 0.05.

On pourra consulter la page https://www.tidymodels.org/find/parsnip/ pour trouver les noms de paramètre du `worfklow` et sur le tutoriel https://juliasilge.com/blog/shelter-animals/ pour la stratégie. Elle est ici de fixer le nombre d'itérations à 500 puisqu'on utilise le **early stopping** en séparant les données en 2. On initialisera donc le workflow avec

```
library(tidymodels)
tune_spec <-
  boost_tree(tree_depth=tune(),trees=500,learn_rate=0.05,
             stop_iter=10) |>
  set_mode("classification") |>
  set_engine("xgboost",validation=0.2)

xgb_wf <- workflow() |>
  add_model(tune_spec) |>
  add_formula(type ~ .)
```

partie IV

Autres

9 Réseaux de neurones avec Keras

Nous présentons ici une introduction au réseau de neurones à l'aide du package **keras**. On pourra trouver une documentation complète ainsi qu'un très bon tutoriel aux adresses suivantes https://keras.rstudio.com et https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/. On commence par charger la librairie

```
library(keras)
#install_keras() 1 seule fois sur la machine
```

Exercice 9.1 (Perceptron avec keras)**.** On va utiliser des réseaux de neurones pour le jeu de données **spam** où le problème est d'expliquer la variable binaire `type`par les 57 autres variables du jeu de données :

```
library(kernlab)
data(spam)
spamX <- as.matrix(spam[,-58])
#spamY <- to_categorical(as.numeric(spam$type)-1, 2)
spamY <- as.numeric(spam$type)-1
```

On sépare les données en un échantillon d'apprentissage et un échantillon test

```
set.seed(5678)
perm <- sample(4601,3000)
appX <- spamX[perm,]
appY <- spamY[perm]
validX <- spamX[-perm,]
validY <- spamY[-perm]
```

1. A l'aide des données d'apprentissage, entraîner un perceptron simple avec une fonction d'activation **sigmoïde**. On utilisera 30 epochs et des batchs de taille 5.

   ```
   #Définition du modèle
   percep.sig <- keras_model_sequential()
   percep.sig %>% layer_dense(units=...,input_shape = ...,activation="...")
   summary(percep.sig)
   percep.sig %>% compile(
     loss="binary_crossentropy",
     optimizer="adam",
     metrics="accuracy"
   )
   ```

```
#Entrainement
p.sig <- percep.sig %>% fit(
  x=...,
  y=...,
  epochs=...,
  batch_size=...,
  validation_split=...,
  verbose=0
)
```

2. Faire de même avec la fonction d'activation **softmax**. On utilisera pour cela 2 neurones avec une sortie Y possédant la forme suivante.

```
spamY1 <- to_categorical(as.numeric(spam$type)-1, 2)
appY1 <- spamY1[perm,]
validY1 <- spamY1[-perm,]
```

3. Comparer les performances des deux perceptrons sur les données de validation à l'aide de la fonction **evaluate**.

4. Construire un ou deux réseaux avec deux couches cachées. On pourra faire varier les nombre de neurones dans ces couches. Comparer les performances des réseaux construits.

Exercice 9.2 (Classification d'images)**.** On considère les données étudiées dans le tutoriel https://tensorflow.rstudio.com/tutorials/keras/classification. On les obtient avec :

```
fashion_mnist <- dataset_fashion_mnist()
c(train_images, train_labels) %<-% fashion_mnist$train
c(test_images, test_labels) %<-% fashion_mnist$test
```

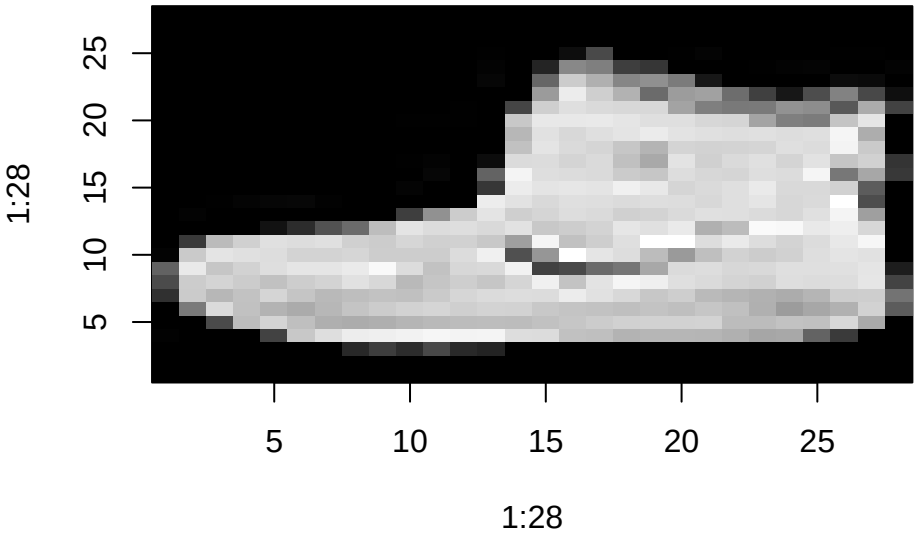
1. Expliquer l'objet `train.images`.

2. Il est d'usage de normaliser les valeurs de pixels :

```
train_images <- train_images / 255
test_images <- test_images / 255
```

On peut alors visualiser une image avec

```
img <- train_images[1,,]
img1 <- t(apply(img, 2, rev))
image(1:28, 1:28, img1, col = gray((0:255)/255))
```



La variable à expliquer est qualitative et prend 10 valeurs :

```
summary(train_labels)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     2.0     4.5     4.5     7.0     9.0
```

Chaque valeur représente le type d'objet associé à l'image. On peut les expliciter avec le vecteur :

```
class_names = c('T-shirt/top',
                'Trouser',
                'Pullover',
                'Dress',
                'Coat',
                'Sandal',
                'Shirt',
                'Sneaker',
                'Bag',
                'Ankle boot')
```

On propose d'utiliser un réseau avec la structure suivante :

```
model <- keras_model_sequential()

model %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(...,...) %>%
  layer_dense(units = 10, activation = 'softmax')
```

La première couche `layer_flatten` est classique lorsqu'on traite des images représentées par des matrices. Elle permet "juste" de tranformer la matrice en un vecteur de dimension

28*28=784. La dernière couche contient 10 neurones et la fonction d'activation **softmax** est définie par :

$$\sigma_i(x_1, \ldots, x_n) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}.$$

Expliquer la pertinence de cette dernière couche ?

3. Entraîner le réseau en choisissant pour la couche cachée :

   - 128 neurones
   - la fonction d'activation `relu`. Puis `'sparse_categorical_crossentropy` comme fonction de perte et 15 epochs pour l'entraînement.

4. Évaluer la performance du réseau sur les données test.

Exercice 9.3 (Comparaison perceptron vs forêt aléatoire avec tidymodels)**.** Toujours pour les données **spam**, on propose ici de comparer le perceptron avec l'algorithme des forêts aléatoires. Pour ce faire on sépare l'échantillon en 2 :

```
library(tidymodels)
set.seed(123)
spam_split <- initial_split(spam,strata=type,prop=3/4)
spam_train <- training(spam_split)
spam_test  <- testing(spam_split)
```

L'échantillon **train** est utilisé pour construire le perceptron et la forêt.

1. Construction du perceptron : la fonction `mlp` de `tidymodels` permet de calibrer différent paramètres du réseau

   ```
   mlp(
     mode = "unknown",
     engine = "nnet",
     hidden_units = NULL,
     penalty = NULL,
     dropout = NULL,
     epochs = NULL,
     activation = NULL,
     learn_rate = NULL
     )
   ```

   On propose d'utiliser par exemple les paramètres suivants où le nombre de neurones dans la couche cachée sera à calibrer :

   ```
   mlp_spec <-
       mlp(penalty = 0, epochs = 100,hidden_units = tune()) %>%
       set_mode("classification") %>%
       set_engine("keras",verbose=FALSE)
   ```

```
mlp_spec
## Single Layer Neural Network Model Specification (classification)
##
## Main Arguments:
##   hidden_units = tune()
##   penalty = 0
##   epochs = 100
##
## Engine-Specific Arguments:
##   verbose = FALSE
##
## Computational engine: keras
```

Construire un workflow pour cet algorithme où sélectionnera le nombre de neurones parmi `c(1,10,50,100)`. On estimera l'accuracy, la balanced accuracy et l'AUC avec une validation hold out pour sélectionner le paramètre :

```
# definition du workflow
mlp_wf <- workflow() |>
  add_model(...) |>
  add_formula(...)
# grille de paramètres
grille_hu <- tibble(...)
# ré-échantillonnage
re_ech_ho <- mc_cv(...)
```

2. Sélectionner une valeur de paramètre et entraîner le modèle sur toutes les données d'apprentissage.
3. Faire le même travail pour une forêt aléatoire avec `ranger`où on sélectionnera le paramètre `mtry` dans `c(1,6,seq(10,50,by=10),57)`.
4. À l'aide de l'échantillon test, comparer les deux algorithmes en terme d'accuracy, de balanced accuracy et d'AUC. On pourra également tracer la courbe ROC.

10 Données déséquilibrées

On parle de **données déséquilibrées** lorsque les deux modalités de la variable cible Y ne sont pas représentées de façon égale dans l'échantillon, ou plus précisément lorsqu'une des deux modalités est fortement majoritaire. Ce contexte est fréquemment rencontré en pratique, on peut citer les cas de détection de fraudes (peu de fraudeurs), de la présence d'une maladie rare (peu de patients atteints), du risque de crédit (peu de mauvais payeurs)... Les algorithmes standards peuvent être mis en difficultés et de nouvelles stratégies doivent être élaborées. Les stratégies classiques permettant de répondre à ce problème consistent à

- utiliser des critères de performance adaptés au déséquilibre ;
- ré-échantillonner les données pour se rapprocher d'une situation d'équilibre.

Nous présentons ces stratégies à travers quelques exercices.

10.1 Critères de performance pour données déséquilibrées

La notion de **risque** en machine learning est capitale puisque c'est à partir de l'estimation de ces risques que l'on **calibre des algorithmes** et que l'on **choisit un algorithme de prévision**. En présence de données déséquilibré, il convient de choisir un risque adapté. En effet, il est le plus souvent important de parvenir à bien identifier des individus de la classe minoritaire. Des critères tels que l'accuracy ou l'erreur de classification ne sont pas pertinents pour ce cadre. On va privilégier des critères comme

- le **balanced accuracy**

$$\text{Bal Acc} = \frac{1}{2}\mathbf{P}(g(X) = 1|Y = 1) + \frac{1}{2}\mathbf{P}(g(X) = -1|Y = -1) = \frac{\text{TPR+TNR}}{2}.$$

- le F_1-score

$$F_1 = 2\,\frac{\text{Precision } \times \text{Recall}}{\text{Precision } + \text{Recall}},$$

avec

$$\text{Precision} = \mathbf{P}(Y = 1|g(X) = 1) \quad \text{et} \quad \text{Recall} = \mathbf{P}(g(X) = 1|Y = 1).$$

- le **kappa de Cohen**

$$\kappa = \frac{\mathbf{P}(a) - \mathbf{P}(e)}{1 - \mathbf{P}(e)}$$

où $\mathbf{P}(a)$ représente l'accuracy et $\mathbf{P}(e)$ l'accuracy sous une hypothèse d'indépendance.

- la courbe ROC et l'AUC...

Comme d'habitude, ces critères sont inconnus et doivent être estimés par des méthodes de ré-échantillonnage de type validation croisée.

Exercice 10.1 (Calculer des critères)**.**

1. Générer un vecteur d'observations **Y** de taille 500 selon une loi de Bernoulli de paramètre 0.05.
2. Générer un vecteur de prévisions **P1** de taille 500 selon une loi de Bernoulli de paramètre 0.01.
3. Générer un vecteur de prévision **P2** de taille 500 tel que

$$\mathcal{L}(P2|Y=0) = \mathcal{B}(0.10) \quad \text{et} \quad \mathcal{L}(P2|Y=1) = \mathcal{B}(0.85).$$

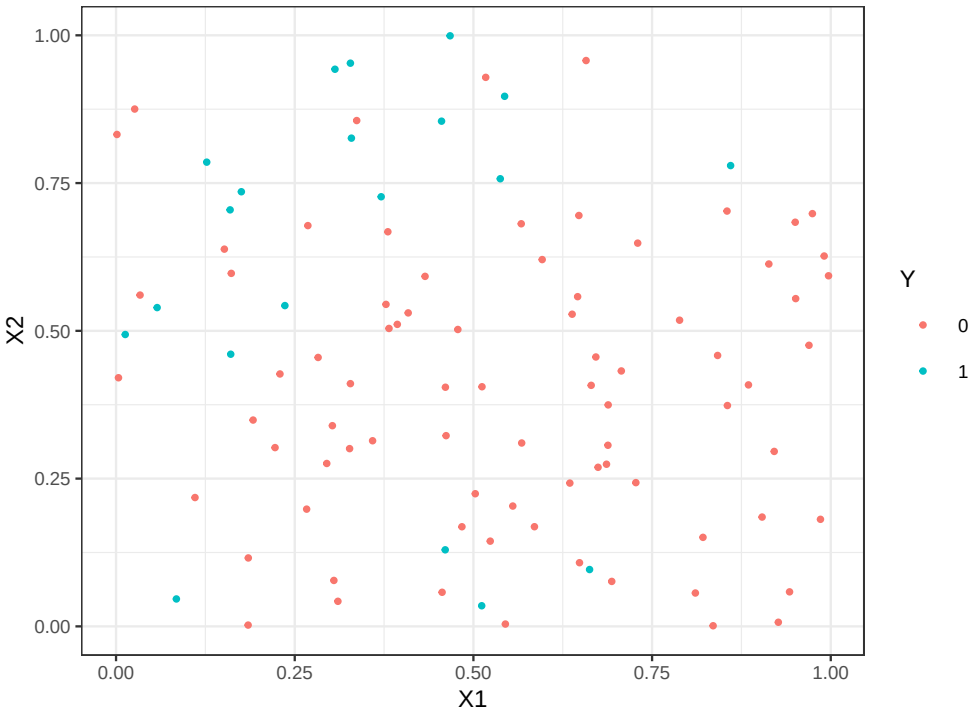
4. Dresser les tables de contingence de **P1** et **P2** à l'aide de **table**. Commenter.
5. Pour **P2**, calculer, avec les fonctions usuelles de `R`, l'**accuracy**, le **recall** et la **précision**.
6. En déduire le F1-score.
7. Même question pour le κ de Cohen.
8. Retrouver ces indicateurs à l'aide des fonctions de package `yardstick` (voir https://yardstick.tidymodels.org/articles/metric-types.html#metrics). On pourra notamment utiliser la fonction **metric_set**. Utiliser également la fonction **confusionMatrix** de **caret**. Analyser les prévisions **P1** et **P2**.

10.2 Ré-équilibrage

En complément du choix d'un **critère pertinent**, il peut être intéressant de tenter de **ré-équilibrer** l'échantillon pour aider les algorithmes à mieux **détecter les individus de la classe minoritaire**. Les méthodes classiques consistent à créer de nouvelles observations de la classe minoritaire (**oversampling**) et/ou supprimer des individus de la classe minoritaire (**undersampling**).

Exercice 10.2 (Quelques algorithmes de ré-équilibrage)**.** On considère le jeu de données `df` ci-dessous où on cherche à prédire `Y` par `X1` et `X2`.

```r
n <- 2000
set.seed(1234)
X1 <- runif(n)
set.seed(5678)
X2 <- runif(n)
set.seed(9012)
R1 <- X1<=0.25
R2 <- (X1>0.25 & X2>=0.75)
R3 <- (X1>0.25 & X2<0.75)
Y <- rep(0,n)
Y[R1] <- rbinom(sum(R1),1,0.75)
Y[R2] <- rbinom(sum(R2),1,0.75)
Y[R3] <- rbinom(sum(R3),1,0.25)
df1 <- tibble(X1,X2,Y)
df1$Y <- factor(df1$Y)
indDY1 <- which(df1$Y==1)
df1.1 <- df1[-indDY1[1:650],]
df1.2 <- df1.1[sample(nrow(df1.1),1000),]
df <- df1.2[sample(nrow(df1.2),100),]
rownames(df) <- NULL
p1 <- ggplot(df)+aes(x=X1,y=X2,color=Y)+geom_point()
p1
```



On a ici 4 fois plus d'observations dans le groupe 0.

```r
summary(df$Y)
```

```
##  0  1
## 80 20
```

1. On commence par faire du **oversampling** avec la fonction **step_upsample** du package `themis`. On pourra s'inspire de https://github.com/tidymodels/themis.

   a. Effectuer le ré-échantillonnage et expliquer.

   b. Corriger les paramètres de la fonction de manière à avoir 80 observations dans le groupe 0 et 60 dans le groupe 1.

2. On s'intéresse maintenant à l'algorithme **SMOTE**

   a. Utiliser la fonction **step_smote** avec `k=3` et les autres valeurs de paramètres par défaut

   b. Visualiser les **observations smote**.

   c. Corriger les paramètres de la fonction de manière à avoir 80 observations dans le groupe 0 et 60 dans le groupe 1.

3. Refaire la question précedente avec l'algorithme ROSE (voir https://journal.r-project.org/archive/2014/RJ-2014-008/RJ-2014-008.pdf).

4. On souhaite maintenant ré-équilibrer par **random undersampling**. Utiliser la fonction **step_downsample** pour effectuer un tel ré-équilibrage. Ici encore on pourra faire varier les paramètres.

5. On passe maintenant à l'algorithme **Tomek**.

   a. Sans utiliser la fonction **step_tomek** identifier les paires d'observations qui ont un **lien de Tomek**. On pourra utiliser la fonction **nng** du package `cccd`.

   b. Retrouver ces paires à l'aide de la fonction **step_tomek**.

   c. Visualiser les observations supprimées.

Exercice 10.3 (Comparer des méthodes de ré-équilibrage avec tidymodels)**.** On considère les données utilisées dans l'exemple https://www.tidymodels.org/learn/models/sub-sampling/ :

```
imbal_data <-
  read_csv("https://tidymodels.org/learn/models/sub-sampling/imbal_data.csv") |>
  mutate(Class = factor(Class))
dim(imbal_data)
## [1] 1200   16
imbal_data |> count(Class)
## # A tibble: 2 x 2
##   Class      n
##   <fct>  <int>
```

```
## 1 Class1    60
## 2 Class2  1140
```

`Class1` est la classe minoritaire et représente l'évènement d'intérêt.

1. Créer une recette permettant de ré-équilibrer les données avec la méthode `ROSE` (on utilisera les paramètres par défaut).
2. On souhaite appliquer l'algorithme des forêts aléatoires sur les données ré-équilibrées. Paramétrer l'algorithme en `tidymodels`. On utilisera comme valeurs de paramètres : `mtry = 3, min_n = 1,trees = 1000`.
3. Créer un **workflow** qui combine la recette et l'algorithme des forêts aléatoires.
4. Évaluer les performances de la méthode en utilisant une validation croisée 10 blocs. On utilisera comme critère l'`accuracy`, l'index `J de Youden`, le `kappa de Cohen`, et l'`auc`.
5. Comparer ces performances avec l'algorithme sans ré-équilibrage. Interpréter.
6. En vous inspirant de la syntaxe présentée ici https://www.tmwr.org/workflow-sets, comparer les performances de l'algorithme des forêts aléatoires utilisant différent type de ré-équilbrage (recettes), par exemple : oversampling, undersampling, smote, rose et aucun ré-équilibrage. On agrégera toutes ces recettes à l'aide de la fonction `workflow_set`.

Exercice 10.4 (Comparaison de méthodes de ré-équilibrage avec le package UBL)**.** On considère 3 jeux de données `df1`, `df2` et `df3`.

```
n <- 2000
set.seed(12345)
X1 <- runif(n)
set.seed(5678)
X2 <- runif(n)
set.seed(9012)
R1 <- X1<=0.25
R2 <- (X1>0.25 & X2>=0.75)
R3 <- (X1>0.25 & X2<0.75)
Y <- rep(0,n)
Y[R1] <- rbinom(sum(R1),1,0.75)
Y[R2] <- rbinom(sum(R2),1,0.75)
Y[R3] <- rbinom(sum(R3),1,0.25)
df1 <- data.frame(X1,X2,Y)
df1$Y <- factor(df1$Y)
indDY1 <- which(df1$Y==1)
df2 <- df1[-indDY1[1:400],]
df3 <- df1[-indDY1[1:700],]
df1 <- df1[sample(nrow(df1),1000),]
df2 <- df2[sample(nrow(df2),1000),]
df3 <- df3[sample(nrow(df3),1000),]
```

1. Comparer la distribution de $\mathbf{Y}$ pour ces trois jeux de données et visualiser les observations.
2. On sépare ces 3 échantillons en un échantillon d'apprentissage et un échantillon test.

```
set.seed(123)
library(caret)
a1 <- createDataPartition(1:nrow(df1),p=2/3)
a2 <- createDataPartition(1:nrow(df2),p=2/3)
a3 <- createDataPartition(1:nrow(df3),p=2/3)
train1 <- df1[a1$Resample1,]
train2 <- df2[a2$Resample1,]
train3 <- df3[a3$Resample1,]
test1 <- df1[-a1$Resample1,]
test2 <- df2[-a2$Resample1,]
test3 <- df3[-a3$Resample1,]
```

   Ajuster une forêt aléatoire sur les 3 échantillon d'apprentissage, calculer les labels prédits sur les échantillons tests et estimer les différents indicateurs vus en cours à l'aide de **confusionMatrix**.
3. On considère uniquement l'échantillon **df3**. Refaire l'analyse précédente en utilisant des techniques de ré-échantillonnage. On pourra utiliser les fonctions du package `UBL`.

10.3 Exercices supplémentaires

Exercice 10.5 (Echantillonnage rétrospectif)**.** Dans le cadre de l'échantillonnage rétrospectif pour le modèle logistique vu en cours, démontrer la propriété qui lie le modèle logistique initial au modèle ré-équilibré.

Exercice 10.6 (Echantillonnage rétrospectif)**.** Une étude cas/témoins est réalisée pour mesurer l'effet du tabac sur une pathologie. Pour ce faire, on choisit $n_1 = 250$ patients atteints de la pathologie (cas) et $n_0 = 250$ patients sains (témoins). Les résultats de l'étude sont présentés ci-dessous

	Fumeur	Non fumeur
Non malade	48	202
Malade	208	42

1. A partir des données obtenues, estimer à l'aide d'un modèle logistique la probabilité d'être atteint pour un fumeur, puis pour un non fumeur.
2. Comment interpréter ces deux probabilités ? Est-ce qu'elles estiment la probabilité d'être atteint pour un individu quelconque dans la population ?

3. Des études précédentes ont montré que cinq individus sur mille sont atteints par la pathologie dans la population entière. En utilisant la propriété de l'exercice précédent, en déduire les probabilités d'être atteint pour un fumeur et un non fumeur dans la population.

11 Comparaison d'algorithmes

Les chapitres précédents ont présenté plusieurs algorithmes permettant de répondre à un problème posé, le plus souvent de classification supervisée. Se pose bien entendu la question de choisir un unique algorithme. Etant donné un échantillon $\mathcal{D}_n = \{(x_1, y_1), \ldots, (x_n, y_y)\}$ on rappelle qu'un algorithme de prévision est une fonction

$$g : \mathcal{X} \times (\mathcal{X} \times \mathcal{Y})^n \to \mathcal{Y}$$

qui, à une nouvelle observation $x \in \mathcal{X}$ renverra la prévision $g(x, \mathcal{D}_n)$ calculée à partir de l'échantillon $\mathcal{D}_n$. Cette fonction g peut contenir tout un tas d'étapes comme :

- la gestion des données manquantes
- une procédure de choix de variables
- une méthode pour ré-équilibrer les données
- des procédures pour calibrer des paramètres (qui peuvent éventuellement inclure des validations croisées)
- ...

Le machine learning se focalisant sur la capacité d'un algorithme à bien prédire, les stratégies classiques pour choisir un algorithme vont (une fois de plus) consister à évaluer le pouvoir prédictif de chaque algorithme. Il n'y a rien de bien nouveau puisque cela va reposer sur les techniques présentées aux chapitres chapitre 2 :

- choisir un ou plusieurs critères (erreur de classification, AUC, F_1-score...)
- choisir une procédure de ré-échantillonnage pour estimer ce critère (validation hold-out, validation croisée, OOB...).

Nous proposons de développer une stratégie pour choisir un algorithme sur le jeu de données **Internet Advertisements Data Set** disponible sur cette page https://archive.ics.uci.edu/ml/datasets/internet+advertisements. Le problème est d'identifier la présence d'une image publicitaire sur des pages webs. Il comporte

```
ad.data <- read.table("data/ad_data.txt",header=FALSE,sep=",",dec=".",na.strings = "?",strip.white = 
dim(ad.data)
## [1] 3279 1559
```

Ce jeu de données contient 1558 variables explicatives, ces variables contiennent différentes caractériques de la page web (voir le site où sont présentées les données pour plus d'information). La dernière variable est la variable à expliquer, elle vaut `ad.` si présence d'une publicité, `nonad.` sinon.

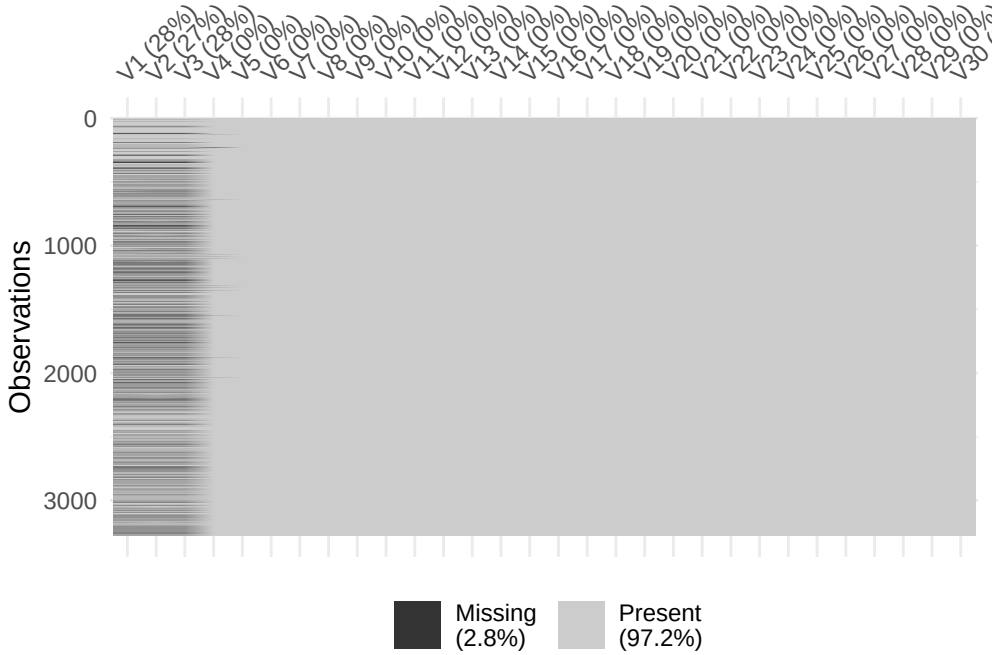
```
names(ad.data)[ncol(ad.data)] <- "Y"
ad.data$Y <- as.factor(ad.data$Y)
summary(ad.data$Y)
##    ad. nonad.
##    459   2820
```

Ce jeu de données contient des données manquantes.

```
sum(is.na(ad.data))
## [1] 2729
```

On peut les visualiser avec

```
library(visdat)
vis_miss(ad.data[,1:30])
```



On remarque que :

- 920 lignes
- 4 colonnes (les 4 premières)

ont au moins une valeur manquante.

```
apply(is.na(ad.data),1,any) %>% sum()
## [1] 920
var.na <- apply(is.na(ad.data),2,any)
names(ad.data)[var.na]
## [1] "V1" "V2" "V3" "V4"
```

On choisit de retirer ces 4 variables de l'analyse (il faudrait peut-être réfléchir un peu plus...).

```
ad.data1 <- ad.data[,var.na==FALSE]
dim(ad.data1)
## [1] 3279 1555
sum(is.na(ad.data1))
## [1] 0
```

On se retrouve donc en présence de 3279 individus et 1554 variables explicatives. On construit la matrice des `X` et le vecteur des `Y` qui sont nécessaires pour certaines fonctions comme `glmnet` :

```
X.ad <- model.matrix(Y~.,data=ad.data1)[,-1]
Y.ad <- ad.data1$Y
```

et on transforme la variable cible en 0-1 pour utiliser `gbm`:

```
ad.data2 <- ad.data1 %>% mutate(Y=recode(Y,"ad."=0,"nonad."=1))
```

On souhaite comparer les algorithmes présentés précédemment. Ils nécessitent les packages suivants

```
library(e1071)
library(caret)
library(rpart)
library(glmnet)
library(ranger)
library(gbm)
```

On commence tout d'abord par représenter un algorithme par une fonction **R** qui admettra en entrée un jeu de données et renverra une unique prévision pour de nouveaux individus. On illustre ces fonctions pour prédire ce nouvel individu.

```
newX <- ad.data1[1000,]
newX.X <- matrix(X.ad[1000,],nrow=1)
```

On stockera les prévisions dans l'objet suivant

```
prev <- tibble(algo=c("SVM","arbre","ridge","lasso","foret","ada","logit"),prev=0)
```

- **SVM** à noyau gaussien où le choix des paramètres du noyau se fait par validation croisée 4 blocs :

```
prev.svm <- function(df,newX){
  C <- c(0.01,1,10)
  sigma <- c(0.1,1,3)
  gr <- expand.grid(C=C,sigma=sigma)
  ctrl <- trainControl(method="cv",number=4)
  cl <- makePSOCKcluster(3)
  registerDoParallel(cl)
  res.svm <- train(Y~.,data=df,method="svmRadial",trControl=ctrl,
                   tuneGrid=gr,prob.model=TRUE)
  stopCluster(cl)
  predict(res.svm,newX,type="prob")[2]
}
prev[1,2] <- prev.svm(ad.data1,newX)
```

- **Arbre de classification** où l'élagage est fait selon la procédure **CART** présentée dans le chapitre 6.

```
prev.arbre <- function(df,newX){
  arbre <- rpart(Y~.,data=df,cp=1e-8,minsplit=2)
  cp_opt <- arbre$cptable %>% as.data.frame() %>% filter(xerror==min(xerror)) %>%
    dplyr::select(CP) %>% slice(1) %>% as.numeric()
  arbre.opt <- prune(arbre,cp=cp_opt)
  predict(arbre,newdata=newX,type="prob")[,2]
}
prev[2,2] <- prev.arbre(ad.data1,newX)
```

- **Lasso et Ridge** où le paramètre de régularisation est choisi par validation croisée 10 blocs en minimisant la déviance binomiale :

```
prev.ridge <- function(df.X,df.Y,newX){
  ridge <- cv.glmnet(df.X,df.Y,family="binomial",alpha=0)
  as.vector(predict(ridge,newx = newX,type="response"))
}
prev.lasso <- function(df.X,df.Y,newX){
  lasso <- cv.glmnet(df.X,df.Y,family="binomial",alpha=1)
  as.vector(predict(lasso,newx = newX,type="response"))
}
prev[3,2] <- prev.ridge(X.ad,Y.ad,newX.X)
prev[4,2] <- prev.lasso(X.ad,Y.ad,newX.X)
```

- **Forêt aléatoire** avec les paramètres par défaut :

```
prev.foret <- function(df,newX){
  foret <- ranger(Y~.,data=df,probability=TRUE)
  predict(foret,data=newX,type="response")$predictions[,2]
}
prev[5,2] <- prev.foret(ad.data1,newX)
```

- **Adaboost et logitboost** avec le nombre d'itérations choisi par validation croisée 5 blocs :

```
prev.ada <- function(df,newX){
  ada <- gbm(Y~.,data=df,distribution="adaboost",interaction.depth=2,
             bag.fraction=1,cv.folds = 5,n.trees=500)
  nb.it <- gbm.perf(ada,plot.it=FALSE)
  predict(ada,newdata=newX,n.trees=nb.it,type="response")
}

prev.logit <- function(df,newX){
  logit <- gbm(Y~.,data=df,distribution="bernoulli",interaction.depth=2,
               bag.fraction=1,cv.folds = 5,n.trees=500)
  nb.it <- gbm.perf(logit,plot.it=FALSE)
  predict(logit,newdata=newX,n.trees=nb.it,type="response")
}

prev[6,2] <- prev.ada(ad.data2,newX)
prev[7,2] <- prev.logit(ad.data2,newX)
```

On peut visualiser la prévision de chaque algorithme

```
prev
## # A tibble: 7 x 2
##   algo   prev
##   <chr> <dbl>
## 1 SVM       0
## 2 arbre     0
## 3 ridge     0
## 4 lasso     0
## 5 foret     0
## 6 ada       0
## 7 logit     0
```

Exercice 11.1 (Choix d'un algorithme par validation croisée)**.** Choisir un algorithme parmi les précédents en utilisant comme critère l'erreur de classification ainsi que la courbe ROC et l'AUC. On pourra faire une validation croisée 10 blocs (même si ça peut être un peu long...).

On ajoute à cette matrice score les valeurs observées que l'on recode en 0-1 :

Exercice 11.2 (Choix d'un algorithme de ré-équilibrage par validation croisée)**.** On considère le même jeu de données que précédemment. Choisir un algorithme de ré-équilibrage par validation croisée. Il s'agira de combiner des méthodes de ré-équilibrage (random over/under sampling, smote, tomek...) avec des algorithmes de prévision de machine learning. On pourra se restreindre au modèle logistique avec calcul des estimateurs par maximum de vraisemblance, ridge, lasso...

Références

Boehmke, B., et B. Greenwell. 2019. *Hands-On Machine Learning with R*. CRC Press. https://bradleyboehmke.github.io/HOML/.

Breiman, L. 2001. « Random forests ». *Machine learning* 45: 5-32.

Breiman, L., J. Friedman, R. Olshen, et C. Stone. 1984. *Classification and regression trees*. Wadsworth & Brooks.

Chen, T., et C. Guestrin. 2016. « XGBoost: A Scalable Tree Boosting System ». In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785-94. KDD '16. New York, NY, USA: ACM. https://doi.org/10.1145/2939672.2939785.

Friedman, J. H. 2001. « Greedy Function Approximation: A Gradient Boosting Machine ». *Annals of Statistics* 29: 1189-1232.

Hastie, T., R. Tibshirani, et J. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer. https://web.stanford.edu/~hastie/ElemStatLearn/.

Ridgeway, G. 2006. « Generalized boosted models: A guide to the gbm package ».