

Machine learning

Laurent Rouvière

2020-10-25

Table des matières

Présentation	1
1 Estimation du risque avec caret	2
1.1 Notion de risque en apprentissage supervisé	2
1.2 La validation croisée	2
1.3 Le package caret	4
1.4 Compléments	6
2 Support Vector Machine (SVM)	8
2.1 Cas séparable	8
2.2 Cas non séparable	10
2.3 L'astuce du noyau	11
2.4 Support vector régression	15
2.5 Exercices	16
3 Arbres	18
3.1 Coupures CART en fonction de la nature des variables	18
3.2 Élagage	21
4 Agrégation : forêts aléatoires et gradient boosting	25
4.1 Forêts aléatoires	25
4.2 Gradient boosting	26
5 Réseaux de neurones avec Keras	28

Présentation

Ce tutoriel présente une introduction au machine learning avec **R**. On pourra trouver :

- les supports de cours associés à ce tutoriel ainsi que les données utilisées à l'adresse suivante https://lrouviere.github.io/machine_learning/ ;
- le tutoriel sans les correction à l'url https://lrouviere.github.io/TUTO_ML/
- le tutoriel avec les corrigés (à certains moment) à l'url https://lrouviere.github.io/TUTO_ML/correction/.

Il est recommandé d'utiliser **mozilla firefox** pour lire le tutoriel.

Les thèmes suivants sont abordés :

- **Estimation du risque**, présentation du package **caret** ;
- **SVM**, cas séparable, non séparable et astuce du noyau ;

- **Arbres**, notamment l'algorithme CART ;
- **Agrégation d'arbres**, forêts aléatoires et gradient boosting ;
- **Réseaux de neurones et introduction au deep learning**, perceptron multicouches avec `keras`.

Il existe de nombreuses références sur le machine learning, la plus connue étant certainement [Hastie et al. \(2009\)](#), disponible en ligne à l'url <https://web.stanford.edu/~hastie/ElemStatLearn/>. On pourra également consulter [Boehmke and 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/>.

1 Estimation du risque avec caret

1.1 Notion de risque en apprentissage supervisé

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

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

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 définir la notion de **critères** que l'on va chercher à optimiser. Les critères sont le plus souvent définis à partir de la fonction de perte

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

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 $(x_1, y_1), \dots, (x_n, y_n)$.

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

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

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 **caret** qui permet de calculer des risques pour quasiment tous les algorithmes que l'on retrouvera en apprentissage supervisé.

1.2 La validation croisée

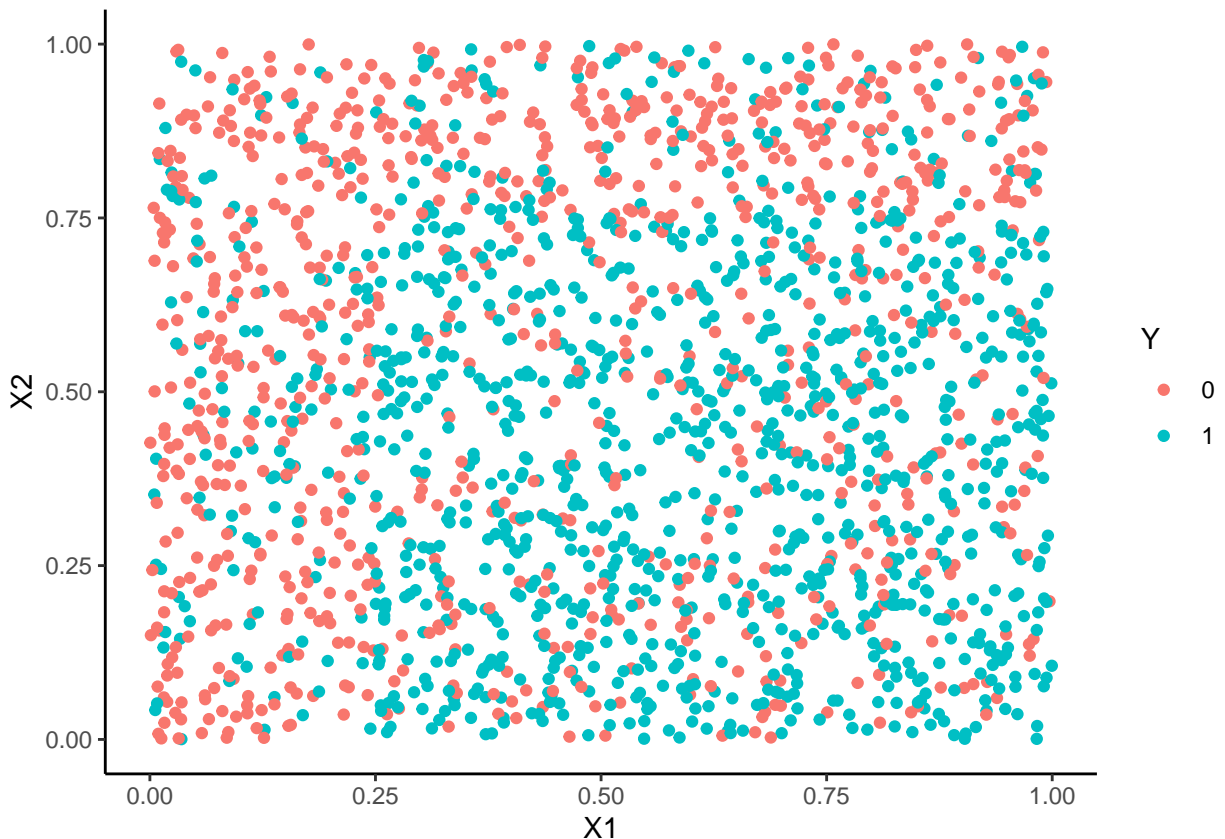
On cherche à expliquer une variable binaire Y par deux variables quantitatives X_1 et X_2 à l'aide du jeu de données suivant

```
n <- 2000
set.seed(12345)
X1 <- runif(n)
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),1,0.25)
Y[R2] <- rbinom(sum(R2),1,0.25)
Y[R3] <- rbinom(sum(R3),1,0.75)
donnees <- data.frame(X1,X2,Y)
donnees$Y <- as.factor(donnees$Y)
ggplot(donnees)+aes(x=X1,y=X2,color=Y)+geom_point()

```



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**.

1. Séparer le jeu de données en un échantillon d'apprentissage **dapp** de taille 1500 et un échantillon test **dtest** de taille 500.
2. On considère la règle de classification des k plus proches voisins. Pour un entier k plus petit que n et un nouvel individu x , cette règle affecte à x le label majoritaire des k plus proches voisins de x . Sur **R** on utilise la fonction **knn** du package **class**. On peut par exemple obtenir les prévisions des individus de l'échantillon test de la règle des 3 plus proches voisins avec

```

library(class)
knn3 <- knn(dapp[,1:2], dtest[,1:2], cl=dapp$Y, k=3)

```

Calculer l'erreur de classification de la règle des 3 plus proches voisins sur les données test (procédure **validation hold out**).

3. Expliquer la fonction **knn.cv**.
4. Calculer l'erreur de classification de la règle des 3 plus proches voisins par validation croisée **leave-one-out**.
5. On considère le vecteur de plus proches voisins suivant :

```
K_cand <- seq(1,500,by=20)
```

Sélectionner une valeur de k dans ce vecteur à l'aide d'une **validation hold out** et d'un **leave-one-out** :

- On calcule l'erreur de classification par **validation hold out** pour chaque valeur de k :

```
err.ho <- rep(0,length(K_cand))
for (i in 1:length(K_cand)){
  ...
  ...
}
```

- On de même chose avec la **validation croisée leave-one-out** :

```
err.loo <- rep(0,length(K_cand))
for (i in 1:length(K_cand)){
  ...
  ...
}
```

6. Faire la même chose à l'aide d'une validation croisée 10 blocs. On pourra construire les blocs avec

```
set.seed(2345)
```

```
blocs <- caret::createFolds(1:nrow(donnees),10,returnTrain = TRUE)
```

```
err.cv <- rep(0,length(K_cand))
prev <- donnees$Y
for (i in 1:length(K_cand)){
  for (j in 1:length(blocs)){
    ...
    ...
  }
  ...
}
K_cand[which.min(err.cv)]
```

1.3 Le package caret

Dans la partie précédente, nous avons utiliser des méthodes de validation croisée pour sélectionner le nombre de voisins dans l'algorithme des plus proches voisins. L'approche revenait à

- estimer un risque pour une grille de valeurs candidates de k
- choisir la valeur de k qui minimise le risque estimé.

Cette pratique est courante en machine learning : on la retrouve fréquemment pour calibrer les algorithmes. Le protocole est toujours le même, pour un méthode donnée il faut spécifier :

- une grille de valeurs pour les paramètres
- un risque
- un algorithme pour estimer le risque.

Le package **caret** permet d'appliquer ce protocole pour plus de 200 algorithmes machine learning. On pourra trouver une documentation complète à cette url <http://topepo.github.io/caret/index.html>. Deux fonctions sont à utiliser :

- **traincontrol** qui permettra notamment de spécifier l'algorithme pour estimer le risque ainsi que les paramètres de cet algorithme;
- **train** dans laquelle on renseignera les données, la grille de candidats...

On reprend les données de la partie précédente.

1. Expliquer les sorties des commandes

```
library(caret)
set.seed(321)
ctrl1 <- trainControl(method="LGOCV",number=1)
KK <- data.frame(k=K_cand)
caret.ho <- train(Y~.,data=donnees,method="knn",trControl=ctrl1,tuneGrid=KK)
caret.ho
k-Nearest Neighbors
```

```
2000 samples
  2 predictor
  2 classes: '0', '1'
```

No pre-processing

Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)

Summary of sample sizes: 1500

Resampling results across tuning parameters:

k	Accuracy	Kappa
1	0.602	0.1956346
21	0.690	0.3649415
41	0.694	0.3736696
61	0.706	0.3992546
81	0.700	0.3867338
101	0.712	0.4122641
121	0.700	0.3882944
141	0.706	0.4017971
161	0.700	0.3903629
181	0.702	0.3941710
201	0.700	0.3898471
221	0.696	0.3806637
241	0.692	0.3714491
261	0.698	0.3829078
281	0.692	0.3693074
301	0.696	0.3764358
321	0.682	0.3474407
341	0.682	0.3468831
361	0.678	0.3352601
381	0.672	0.3214167
401	0.668	0.3113633
421	0.666	0.3057172
441	0.658	0.2853800
461	0.658	0.2841354
481	0.654	0.2732314

Accuracy was used to select the optimal model using the largest value.

The final value used for the model was k = 101.

```
plot(caret.ho)
```

2. En modifiant les paramètres du code précédent, retrouver les résultats de la validation hold out de la partie précédente. On pourra utiliser l'option `index` dans la fonction `trainControl`.
3. Utiliser `caret` pour sélectionner k par validation croisée leave-one-out.
4. Faire de même pour la validation croisée 10 blocs en gardant les mêmes blocs que dans la partie précédente.

1.4 Compléments

1.4.1 Calcul parallèle

Les validations croisées 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 `caret`, on peut par exemple utiliser la librairie `doParallel` :

```
library(doParallel)
cl <- makePSOCKcluster(1)
registerDoParallel(cl)
system.time(ee3 <- train(Y~.,data=donnees,method="knn",trControl=ctrl4,tuneGrid=KK))
  user  system elapsed
 14.182   0.191  15.849
stopCluster(cl)
cl <- makePSOCKcluster(4)
registerDoParallel(cl)
system.time(ee3 <- train(Y~.,data=donnees,method="knn",trControl=ctrl4,tuneGrid=KK))
  user  system elapsed
  0.611   0.020   7.393
stopCluster(cl)
```

1.4.2 Répéter les méthodes de rééchantillonnage

Les méthodes d'estimation du risque présentées dans cette partie (hold out, validation croisée) sont basées sur du **rééchantillonnage**. 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 **caret**, par exemple pour

— la validation hold out on utilise l'option `number` :

```
ctrl <- trainControl(method="LGOCV",number=5)
caret.ho.rep <- train(Y~.,data=donnees,method="knn",trControl=ctrl,tuneGrid=KK)
```

— la validation croisée on utilise les options `repeatedcv` et `repeats` :

```
ctrl <- trainControl(method="repeatedcv",repeats=5)
caret.ho.rep <- train(Y~.,data=donnees,method="knn",trControl=ctrl,tuneGrid=KK)
```

1.4.3 Modifier le risque

Enfin nous avons uniquement considéré l'erreur de classification. Il est bien entendu possible d'utiliser d'autres **risques** pour évaluer les performances. C'est l'option `metric` de la fonction **train** qui permet généralement de spécifier le risque, si on est par exemple intéressé par l'**aire sur la courbe ROC (AUC)** on fera :

```
donnees1 <- donnees
names(donnees1)[3] <- "Class"
levels(donnees1$Class) <- c("G0","G1")
ctrl <- trainControl(method="LGOCV",number=1,classProbs=TRUE,summary=twoClassSummary)
caret.auc <- train(Class~.,data=donnees1,method="knn",trControl=ctrl,tuneGrid=KK,metric="ROC")
caret.auc
k-Nearest Neighbors
```

```
2000 samples
  2 predictor
  2 classes: 'G0', 'G1'
```

No pre-processing

Resampling: Repeated Train/Test Splits Estimated (1 reps, 75%)

Summary of sample sizes: 1500

Resampling results across tuning parameters:

k	ROC	Sens	Spec
1	0.5904827	0.5758929	0.6050725
21	0.6945765	0.6294643	0.7789855
41	0.7206506	0.6250000	0.7789855
61	0.7291424	0.6250000	0.7753623
81	0.7247752	0.6205357	0.7934783
101	0.7241282	0.6250000	0.7934783
121	0.7204322	0.6250000	0.8007246
141	0.7198580	0.6294643	0.7862319
161	0.7221791	0.6250000	0.7826087
181	0.7225188	0.6205357	0.7826087
201	0.7170597	0.6205357	0.7934783
221	0.7143100	0.6160714	0.7862319
241	0.7196801	0.6205357	0.7898551
261	0.7150055	0.6205357	0.7898551
281	0.7184669	0.6116071	0.7898551
301	0.7187096	0.5892857	0.7971014

321	0.7187904	0.5758929	0.8007246
341	0.7178927	0.5491071	0.8079710
361	0.7158789	0.5178571	0.8224638
381	0.7177957	0.5089286	0.8224638
401	0.7168818	0.4776786	0.8297101
421	0.7174964	0.4598214	0.8478261
441	0.7134770	0.4330357	0.8623188
461	0.7141401	0.4241071	0.8695652
481	0.7131535	0.4107143	0.8659420

ROC was used to select the optimal model using the largest value.
The final value used for the model was k = 61.

2 Support Vector Machine (SVM)

Etant donnée un échantillon $(x_1, y_1), \dots, (x_n, y_n)$ 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**.

2.1 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 :

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

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

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

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

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

$$\text{— } \alpha_i^* = 0$$

ou

$$\text{— } y_i(x_i^t w + b) - 1 = 0.$$

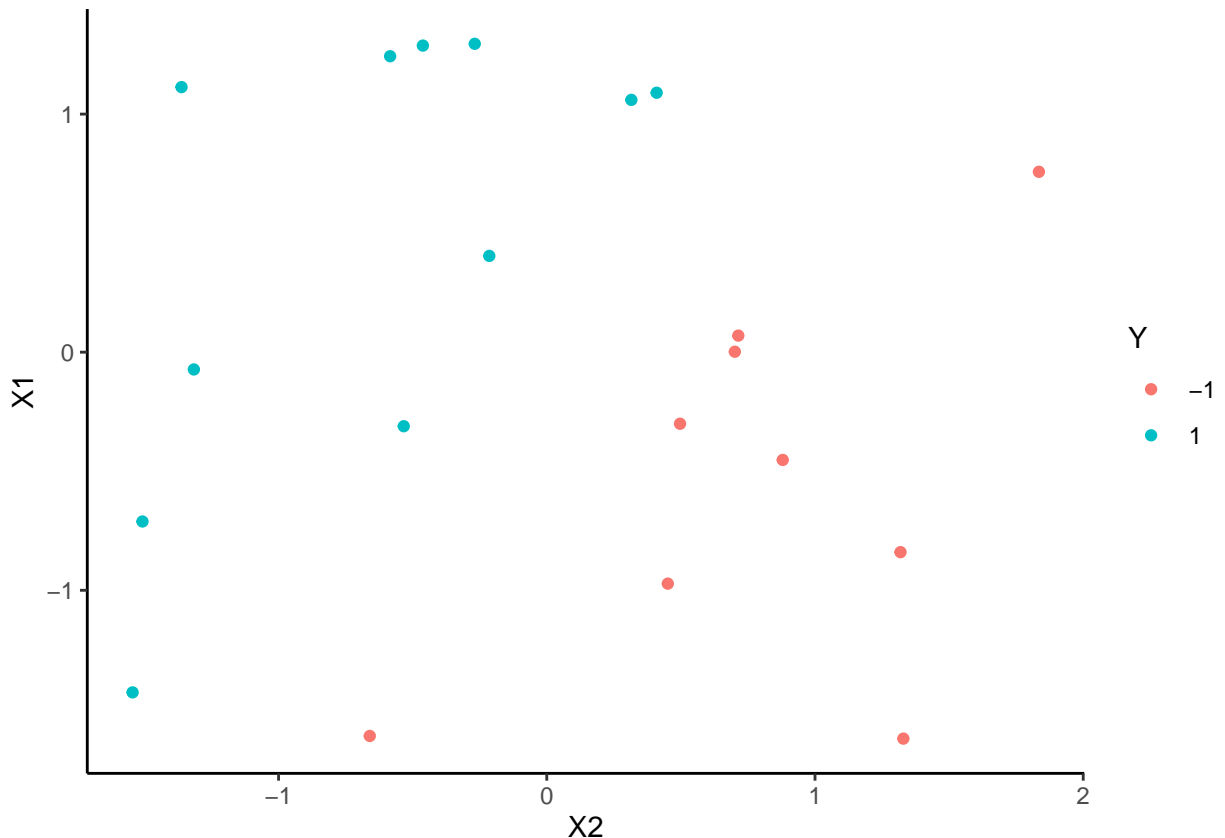
Ces conditions impliquent que w^* s'écrit comme une combinaison linéaire de quelques points, appelés **vecteurs supports** qui se trouvent **sur la marge**. Nous proposons maintenant de retrouver ces points et de tracer la marge sur un exemple simple.

On considère le nuage de points suivant :


```

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

```



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 **data.frame** 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. Retrouver ce graphe à l'aide de la fonction **plot**.
3. 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**.
4. On dispose d'un nouvel individu $x = (-0.5, 0.5)$. Expliquer comment on peut prédire son groupe.

5. Retrouver les résultats de la question précédente à l'aide de la fonction **predict**. On pourra utiliser l'option **decision.values = TRUE**.
6. Obtenir les probabilités prédites à l'aide de la fonction **predict**. On pourra utiliser **probability=TRUE** dans la fonction **svm**.

2.2 Cas non séparable

Dans la vraie vie, les groupes ne sont généralement pas séparables et il n'existe donc pas de solution au problème (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)** ξ_1, \dots, ξ_n positives telles que :

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

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

$$\frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \xi_i$$

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

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

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

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

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

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**).
- $C \nearrow \implies \xi_i \searrow$ 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 1 :

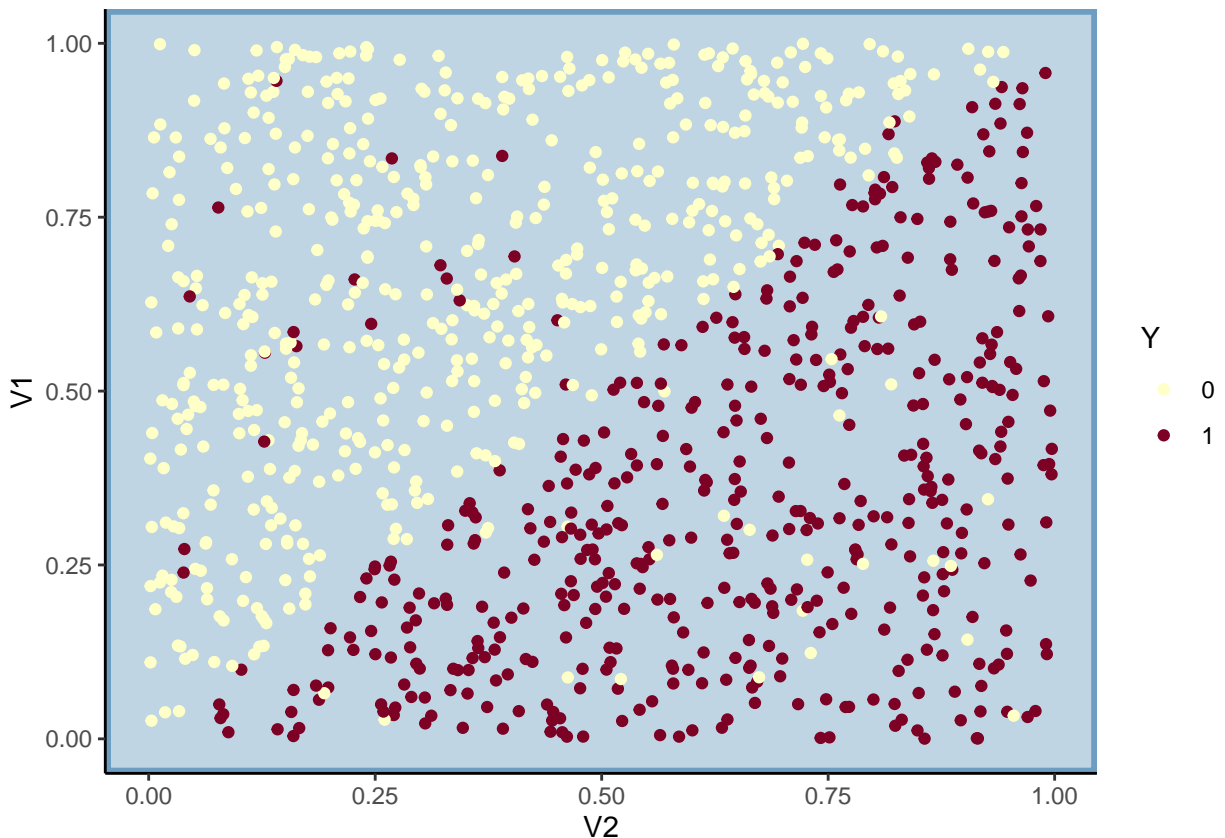
- choix d'une grille de valeurs de C et d'un critère ;
- choix d'une méthode de ré-échantillonnage pour estimer le critère ;
- choix de la valeur de C qui minimise le critère estimé.

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(nrow(.),1,0.95))
df2 <- df %>% filter(V1>V2)%>% mutate(Y=rbinom(nrow(.),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("#FFFC8", "#7D0025"))+
  theme(panel.background = element_rect(fill = "#BFD5E3", colour = "#6D9EC1",size = 2, linetype = "solid"),
        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.

2.3 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

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

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)

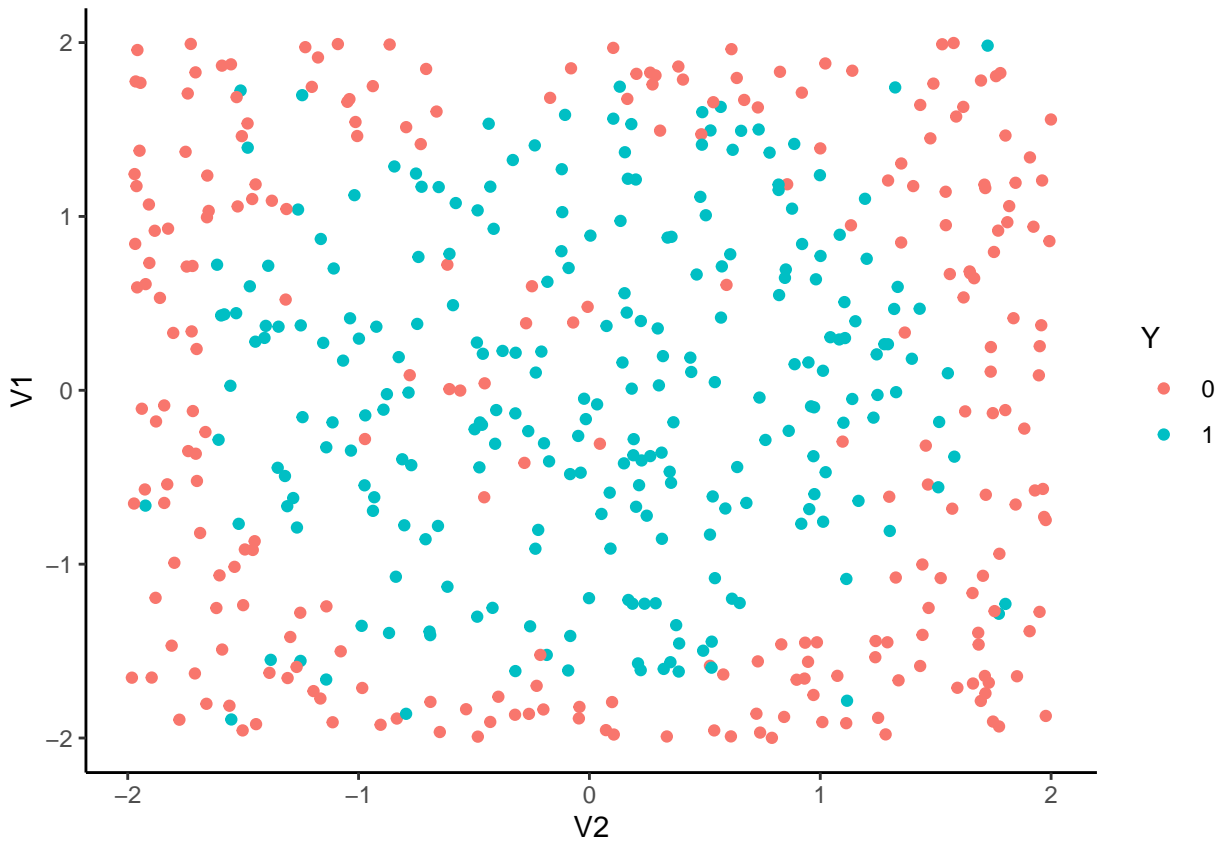
$$K(x, x') = \exp\left(-\frac{\|x - x'\|}{2\sigma^2}\right).$$

- **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** !

On considère le jeu de données suivant où le problème est d'expliquer Y par $V1$ et $V2$.

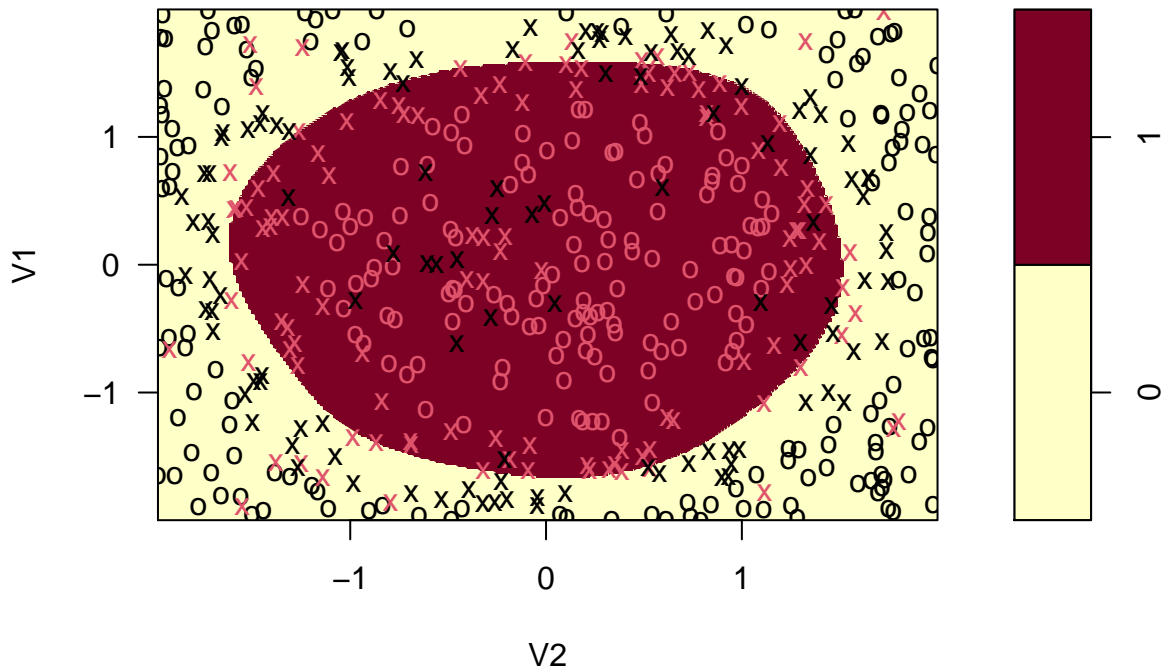
```
n <- 500
set.seed(13)
X <- matrix(runif(n*2,-2,2),ncol=2) %>% as.data.frame()
Y <- rep(0,n)
cond <- (X$V1^2+X$V2^2)<=2.8
Y[cond] <- rbinom(sum(cond),1,0.9)
Y[!cond] <- rbinom(sum(!cond),1,0.1)
df <- X %>% mutate(Y=as.factor(Y))
ggplot(df)+aes(x=V2,y=V1,color=Y)+geom_point()+theme_classic()
```



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)
```

SVM classification plot



3. Faire varier les paramètres **gamma** et **cost**. Interpréter (on pourra notamment étudier l'évolution du nombre de vecteurs supports en fonction du paramètre **cost**).

```
mod.svm2 <- svm(Y~.,data=df,kernel="radial",gamma=...,cost=...)
mod.svm3 <- svm(Y~.,data=df,kernel="radial",gamma=...,cost=...)
mod.svm4 <- svm(Y~.,data=df,kernel="radial",gamma=...,cost=...)

plot(mod.svm2,df,grid=250)
plot(mod.svm3,df,grid=250)
plot(mod.svm4,df,grid=250)

mod.svm2$nSV
mod.svm3$nSV
mod.svm4$nSV
```

4. Sélectionner automatiquement ces paramètres. On pourra utiliser la fonction **tune** en faisant varier **C** dans **c(0.1,1,10,100,1000)** et **gamma** dans **c(0.5,1,2,3,4)**.

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

5. Faire de même avec **caret**, on utilisera **method="svmRadial"** et **prob.model=TRUE**.

```
C <- c(0.001,0.01,1,10,100,1000)
sigma <- c(0.5,1,2,3,4)
gr <- expand.grid(C=C,sigma=sigma)
ctrl <- trainControl(...)
res.caret1 <- train(...,prob.model=TRUE)
res.caret1
```

6. Visualiser la règle sélectionnée.

2.4 Support vector régression

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

$$\frac{1}{2} \|w\|^2$$

$$\text{sous les contraintes } |y_i - \langle w, x_i \rangle - b| \leq \varepsilon, \quad i = 1, \dots, n,$$

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, ξ, ξ^*) qui minimise

$$\frac{1}{2} \|w\|^2 + C \sum_{i=1}^n (\xi_i + \xi_i^*)$$

$$\text{sous les contraintes } \begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i, & i = 1, \dots, n, \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^*, & i = 1, \dots, n \\ \xi_i \geq 0, \xi_i^* \geq 0, & i = 1, \dots, n \end{cases}$$

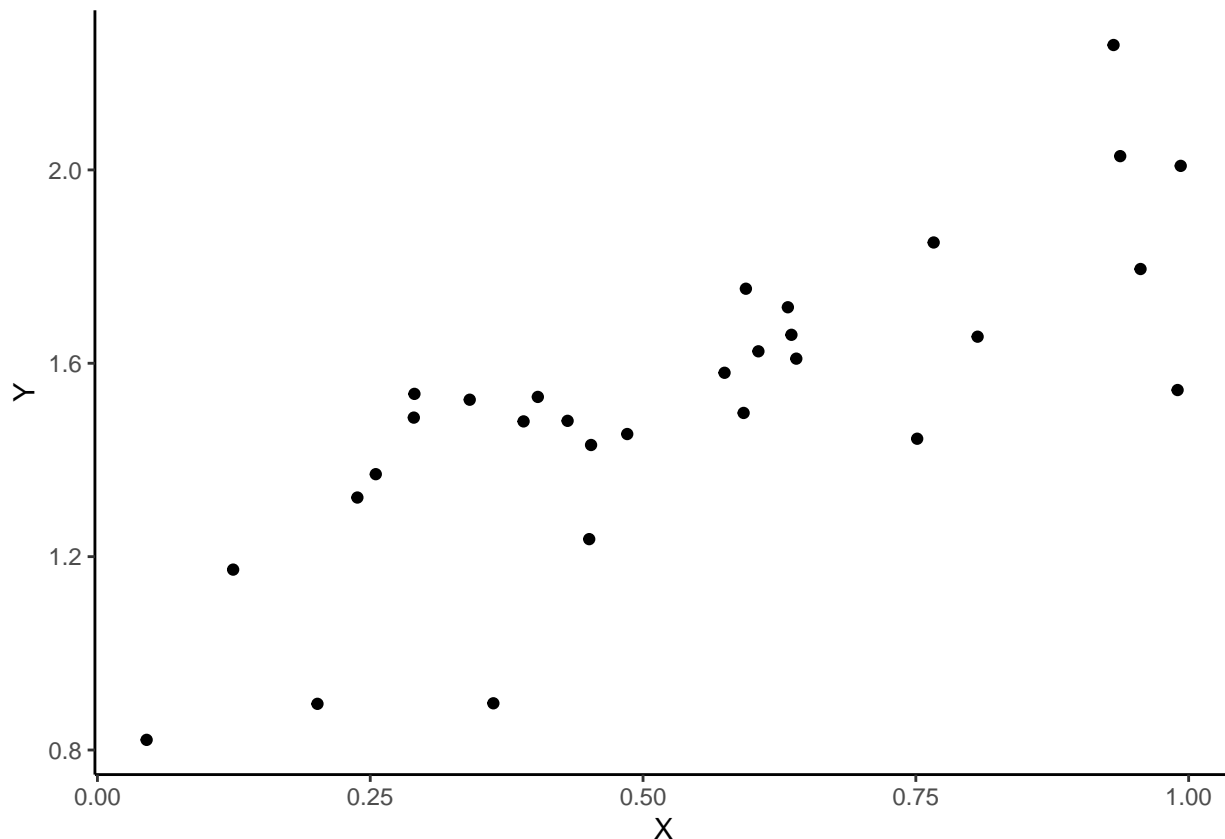
Les solutions s'obtiennent exactement de la même façon que dans le cas binaire. On montre notamment que w^* s'écrit comme une combinaison linéaire de vecteurs supports :

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

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

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

```
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** :

```
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 droite ainsi que la marge.
2. Comparer la **SVR** précédente avec celle utilisant **epsilon=0.7**.
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**.

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

2.5 Exercices

Exercice 2.1 (Résolution du problème d'optimisation dans le cas séparable).

On considère n observations $(x_1, y_1), \dots, (x_n, y_n)$ 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 $x_i, i = 1, \dots, n$ et \mathcal{H} en fonction de w et b .
2. Expliquer la logique du problème d'optimisation

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

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

3. Montrer que ce problème peut se réécrire

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

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

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

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

Si on désigne par $u_\alpha = \operatorname{argmin}_u L(u, \alpha)$, 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 α^* la solution de ce problème, on déduit la solution du problème primal $u^* = u_{\alpha^*}$. Les conditions de Karush-Kuhn-Tucker sont données par

- $\alpha_i^* \geq 0$.
- $g_i(u_{\alpha^*}) \geq 0$.
- $\alpha_i^* g_i(u_{\alpha^*}) = 0$.
- 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^* et b^* .
- d. Interpréter les conditions KKT.

Exercice 2.2 (Règle svm à partir de sorties R).

On considère n observations $(x_1, y_1), \dots, (x_n, y_n)$ 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)

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

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

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

1. Donner la formule permettant de calculer w^* en fonction des α_i^* .
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^* et b^* . 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 ?

3 Arbres

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

3.1 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 $(x_1, y_1), \dots, (x_n, y_n)$ 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

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

où $\bar{y}_{\mathcal{N}}$ 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 nœud père et ses deux nœuds fils :

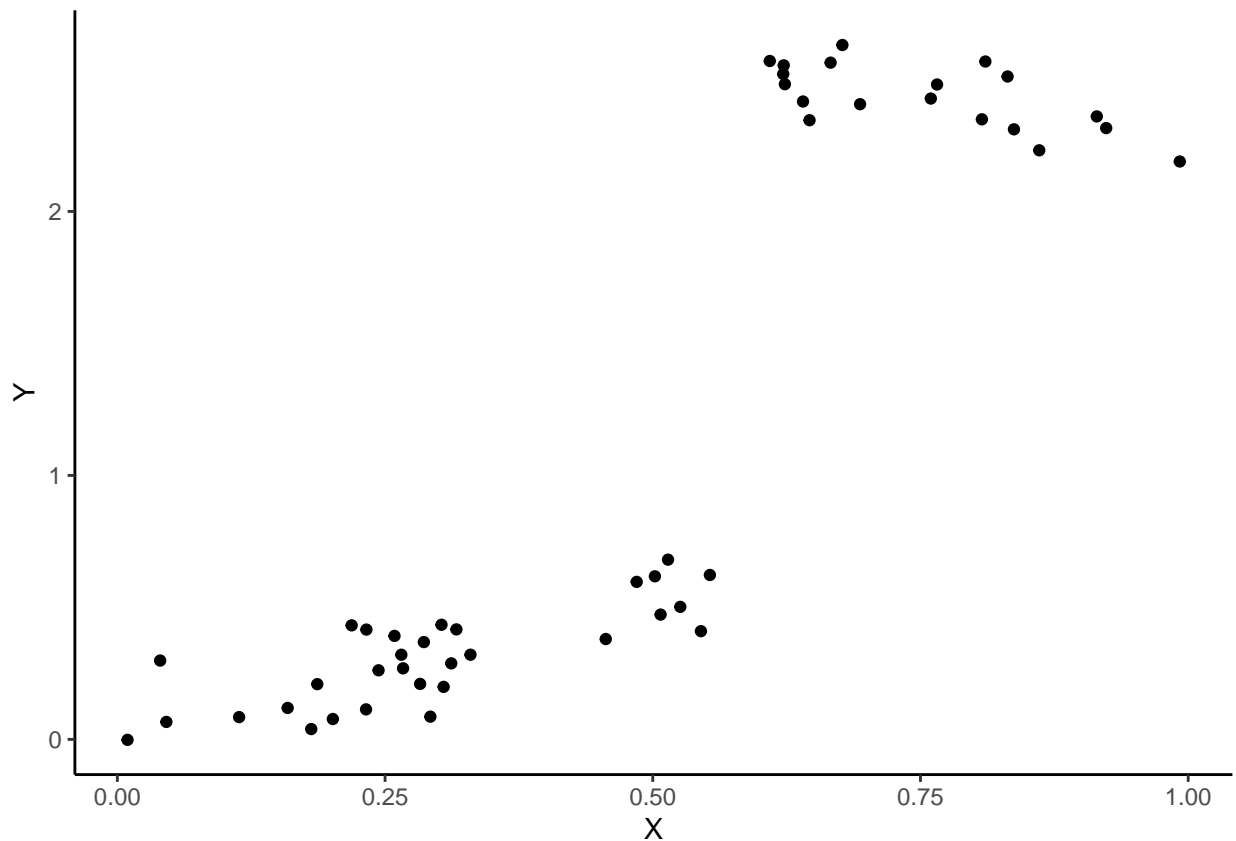
$$\Delta(\mathcal{I}) = \mathbf{P}(\mathcal{N})\mathcal{I}(\mathcal{N}) - (\mathbf{P}(\mathcal{N}_1(j, s))\mathcal{I}(\mathcal{N}_1(j, s)) + \mathbf{P}(\mathcal{N}_2(j, s))\mathcal{I}(\mathcal{N}_2(j, s)))$$

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} .

3.1.1 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()
```

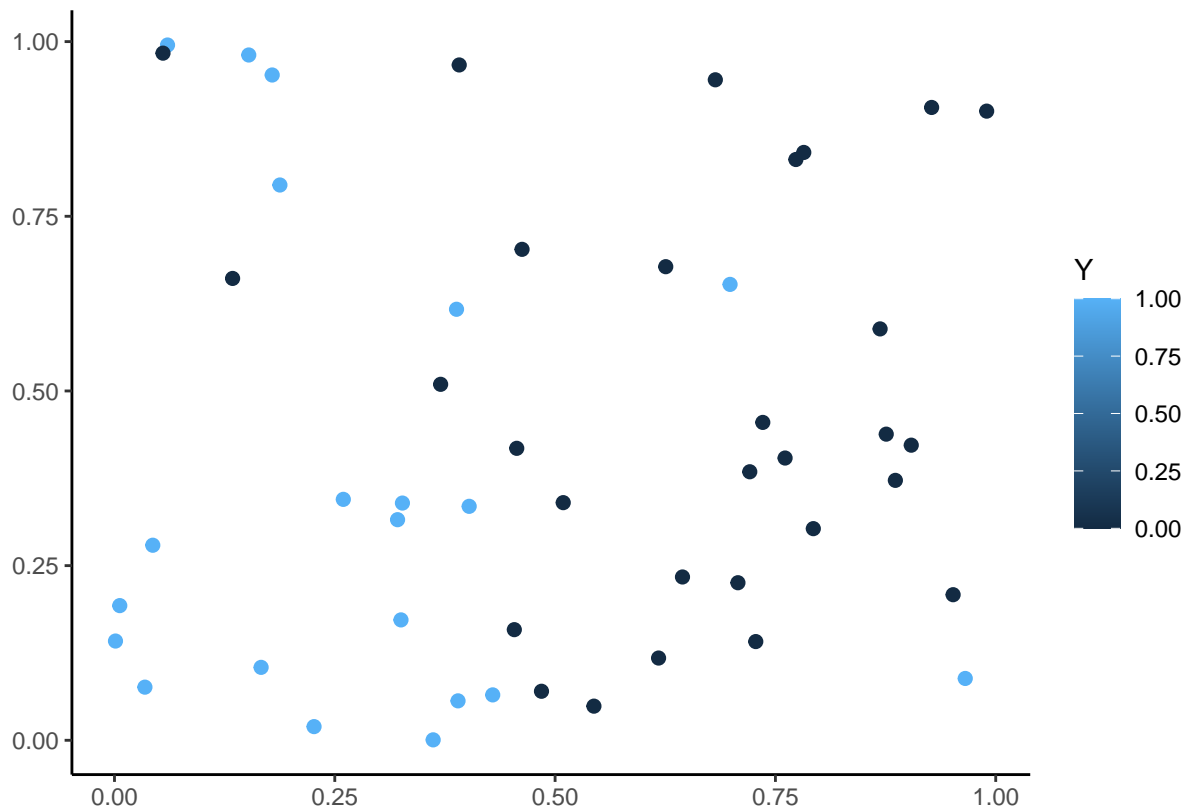


1. A l'aide de la fonction **rpart** du package **rpart**, construire un arbre permettant d'expliquer Y par X .
2. Visualiser l'arbre à l'aide des fonctions **prp** et **rpart.plot** du package **rpart.plot**.
3. Écrire l'estimateur associé à l'arbre.
4. Ajouter sur le graphe de la question 1 la partition définie par l'arbre ainsi que les valeurs prédites.

3.1.2 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="")+theme_classic()
```



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.

3.1.3 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.

3.2 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 emboîtés** :

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

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 \mathcal{T}_{max} , c'est pourquoi on parle **d'élagage**.

3.2.1 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 %>% slice(3,58,185,218) %>% select(-Sales)
new_ind
```

	CompPrice	Income	Advertising	Population	Price	ShelveLoc
3	113	35	10	269	80	Medium
58	93	91	0	22	117	Bad
185	132	33	7	35	97	Medium
218	106	44	0	481	111	Medium

	Age	Education	Urban	US
3	59	12	Yes	Yes
58	75	11	Yes	No
185	60	11	No	Yes
218	70	14	No	No

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.

3.2.2 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,minsplitt=2)

```

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

```

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, minsplitt = 2)

```

```

Variables actually used in tree construction:
[1] Advertising Age          CompPrice  Education
[5] Income      Population Price        ShelfLoc

```

Root node error: 236/400 = 0.59

n= 400

	CP	nsplitt	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.

4 Agrégation : forêts aléatoires et gradient boosting

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, \dots, 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.

4.1 Forêts aléatoires

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
- $m \in \{1, \dots, d\}$ le nombre de variables candidates pour découper un nœud.

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

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**. Nous la présentons à travers l'exemple du jeu de données **spam** du package **kernlab**.

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

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

1. A l'aide de la fonction **randomForest** du package **randomForest**, ajuster une forêt aléatoire pour répondre au problème posé.
2. Appliquer la fonction **plot** à l'objet construit avec **randomForest** et expliquer le graphe obtenu. A quoi peut servir ce graphe en pratique ?
3. Construire la forêt avec **mtry=1** et comparer ses performances avec celle construite précédemment.
4. Utiliser la fonction **train** du package **caret** pour choisir le paramètre **mtry** dans la grille **seq(1,30,by=5)**.
5. Construire la forêt avec le paramètre **mtry** sélectionné. Calculer l'importance des variables et représenter ces importance à l'aide d'un diagramme en barres.

6. La fonction **ranger** du package **ranger** permet également de calculer des forêts aléatoires. Comparer les temps de calcul de cette fonction avec **randomForest**

4.2 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 $\ell : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ 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 :

- $d_n = (x_1, y_1), \dots, (x_n, y_n)$ 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 : $g_0(\cdot) = \operatorname{argmin}_c \frac{1}{n} \sum_{i=1}^n \ell(y_i, c)$
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 $g_{m-1}(x_i)$:

$$U_i = -\frac{\partial}{\partial g(x_i)} \ell(y_i, g(x_i)) \Big|_{g(x_i)=g_{m-1}(x_i)}, \quad i = 1, \dots, n.$$
 - b. Ajuster un arbre sur l'échantillon $(x_1, U_1), \dots, (x_n, U_n)$, 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](#)).

4.2.1 Un exemple simple en régression

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

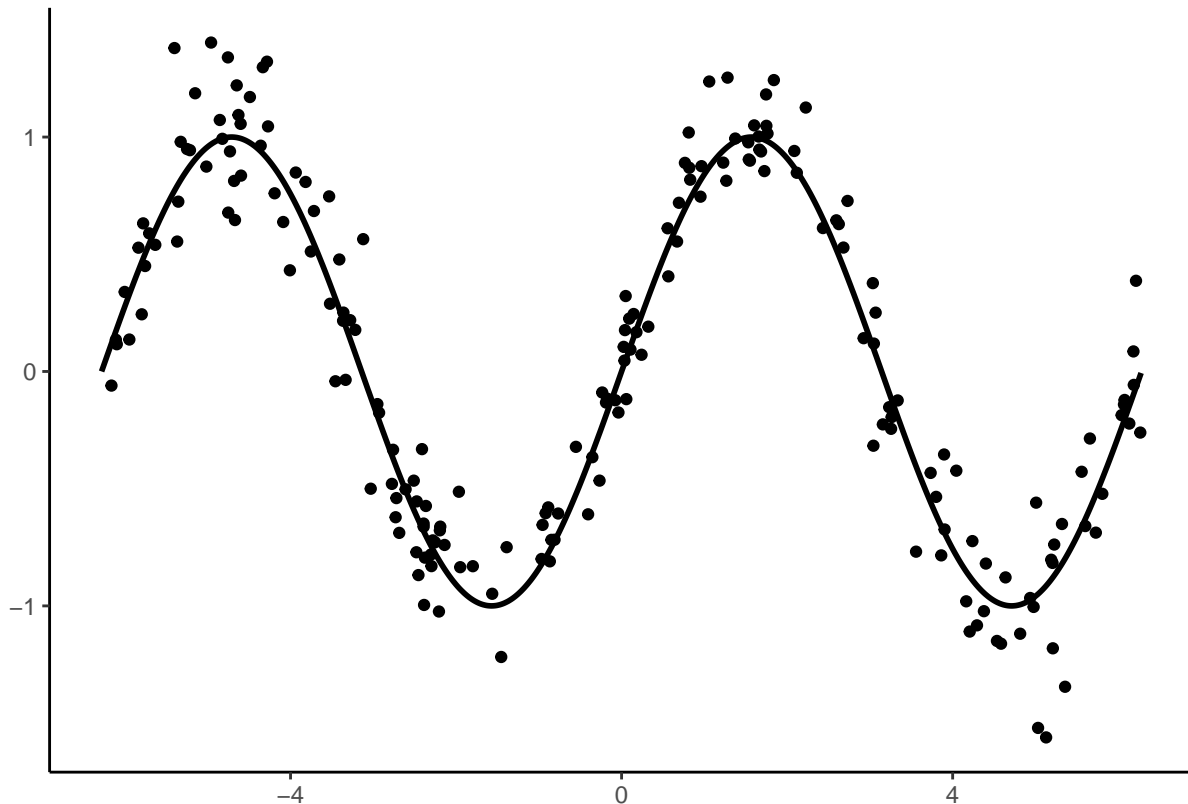
$$y_i = m(x_i) + \varepsilon_i$$

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 <- data.frame(X,Y)
df2 <- data.frame(X=x,Y=y)
p1 <- ggplot(df1)+aes(x=X,y=Y)+geom_point()+geom_line(data=df2,size=1)+xlab("")+ylab("")
p1

```



1. Rappeler ce que signifie 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.
3. Visualiser l'estimateur à la première itération. On pourra faire un **predict** avec l'option **n.trees**.
4. Faire de même pour les itérations 1000 et 500000.
5. Sélectionner le nombre d'itérations par la procédure de votre choix.

4.2.2 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

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

2. Proposer une correction permettant de faire fonctionner l'algorithme.
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 de la validation croisée 5 blocs.
7. Faire la même procédure en changeant la valeur du paramètre **shrinkage**. Interpréter.
8. Expliquer la différence entre **adaboost** et **logitboost** et précisez comment on peut mettre en œuvre ce dernier algorithme.

5 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
```

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, entrainer 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"
)
#Entraînement
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.

Références

- Boehmke, B. and Greenwell, B. (2019). *Hands-On Machine Learning with R*. CRC Press.
- Breiman, L. (2001). Random forests. *Machine learning*, 45 :5–32.
- Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and regression trees*. Wadsworth & Brooks.
- Friedman, J. H. (2001). Greedy function approximation : A gradient boosting machine. *Annals of Statistics*, 29 :1189–1232.
- Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning : Data Mining, Inference, and Prediction*. Springer, second edition.
- Ridgeway, G. (2006). Generalized boosted models : A guide to the gbm package.