Module 5 - Prédiction et Classification

Dans ce module, nous allons aborder d'autres méthodes de régression et classification que celles vues précédemment, davantage orientées prédiction. À cette occasion, nous verrons quelques enjeux et concepts de machine learning.

1 Introduction au machine learning

Le machine learning correspond à un domaine scientifique qui met en oeuvre des algorithmes qui apprennent sur les données qu'on leur fournit. La frontière entre la modélisation statistique et le machine learning est poreuse, les disciplines empruntent l'une à l'autre.
En machine learning, on distingue généralement deux grands types de problématiques :

1. L'**apprentissage non supervisé** = trouver une « structure » dans des données non-labellisées
   $\rightarrow$ clustering, réduction de dimension (cf. module 3)
   $\rightarrow$ *subjectif (il n'y a pas de vérité), mais utile à titre exploratoire ou en étape de pré-traitement*
2. L'**apprentissage supervisé** = inférer (prédire) une fonction ou une relation à partir de données d'apprentissage labellisées
   $\rightarrow$ régressions, classification (cf. module 4)

1.1 L'apprentissage supervisé

Formellement, un problème d'apprentissage supervisé revient à trouver une fonction f qui associe les variables en entrée X et les variables en sortie Y :

$$Y = f(X)$$

On distingue 2 cas :

1. les modèles paramétriques où on impose une forme à f
   par exemple : $f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2$
   Les + : simple, rapide
   Les - : pas toujours adapté aux données
2. les modèles non paramétriques : aucune hypothèse sur f
   Les + : flexible, performant
   Les - : demande une grande volumétrie de données, risque de sur-apprentissage

L'objectif est de **généraliser l'information** contenue dans les données pour **l'appliquer sur de nouvelles données** pour réaliser une prédiction.
Lorsqu'un modèle ne généralise pas suffisamment, on parle de **sur-apprentissage**. Pour éviter cela, on peut mettre en uvre plusieurs **stratégies d'échantillonnage** : on entraîne le modèle sur un échantillon des données et on l'évalue sur un autre échantillon indépendant.

1.2 Le sur-apprentissage

Le biais et la variance

Un modèle avec un faible biais (ou *erreur d'approximation*) fera peu d'erreur de prédiction sur l'échantillon. Un modèle avec une faible variance (ou *erreur d'estimation*) aura une prédiction relativement stable, peu fluctuante. En machine learning, on ne peut pas avoir les deux ! En effet, ajouter des variables/des paramètres (*complexifier le modèle*) permet de s'approcher des données (moins de biais) mais augmente dans le même temps la volatilité de la prédiction (plus de variance).

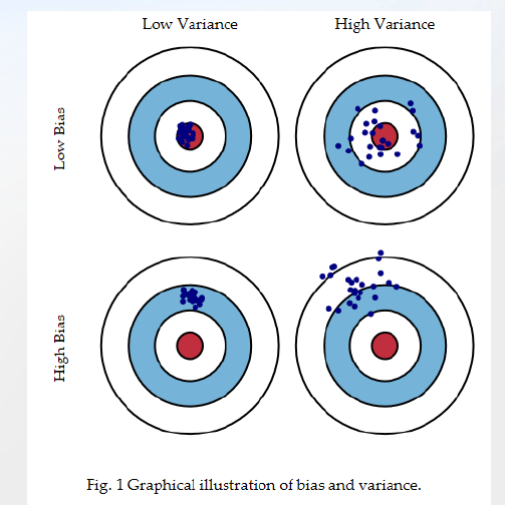

FIGURE 1 – Le biais et la variance

Le compromis biais-variance

Le compromis biais-variance est un concept fondamental en statistique et machine learning. On peut le décrire de la façon suivante :

— un modèle doit *être assez complexe* pour bien expliquer les données d'un échantillon d'entraînement (et donc avoir un biais relativement faible). En effet, s'il est trop simple et

n'explique pas bien les données d'entraînement, il n'expliquera pas correctement les données de test : on parle de sous-apprentissage.

— un modèle ne doit être *pas être trop complexe* (avoir trop de paramètres par exemple, qui résolvent des cas particuliers) car le modèle sera trop centré sur les données d'entraînement et impliquera une grande variance. En schématisant, le modèle serait fait sur-mesure au niveau de l'échantillon d'entraînement et sera donc peu performant sur l'échantillon de test : on parle dans ce cas de sur-apprentissage.

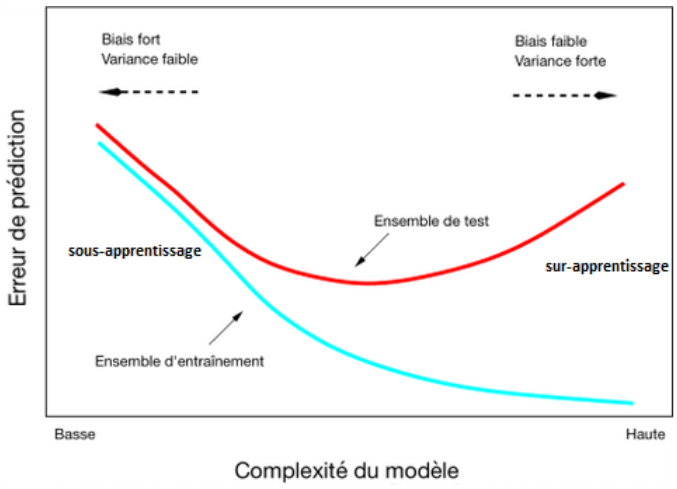

FIGURE 2 – Le compromis biais-variance

→ En machine learning, l'important est de trouver un juste milieu ! On recherche un **compromis entre l'adéquation aux données et la complexité du modèle** pour pouvoir ensuite le généraliser.

Quel modèle choisir ?

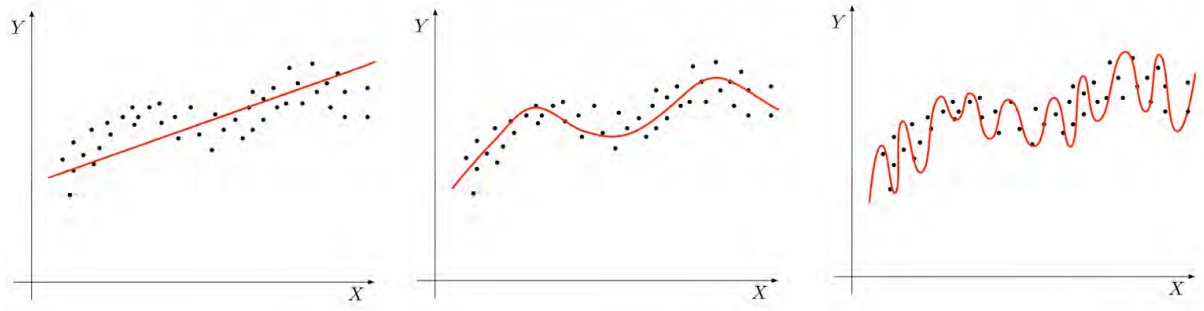

FIGURE 3 – Des modèles de complexité croissante

L'échantillonnage par validation croisée

Une stratégie possible pour complexifier le modèle tout en évitant le sur-apprentissage consiste à ré-échantillonner les données en plusieurs blocs. La **validation croisée** consiste à entraîner le modèle sur une fraction des données (l'échantillon *train*) puis à l'évaluer sur une autre (l'échantillon *test*), en faisant varier ces fractions train et test.
Formellement :

1. Diviser aléatoirement les données en K blocs égaux.
   Le bloc k contient n_k observations : $n_k = n/K$
2. Pour $k \in \{1, ..., K\}$:
   (a) Retirer le bloc k de la base d'apprentissage
   (b) Estimer la fonction de prévision sur la base d'apprentissage (= entraîner le modèle)
   (c) Calculer un critère d'erreur de prévision sur le bloc k : CV_k (par exemple, le taux de mal classés ou la somme des erreurs au carré)
3. Calculer le critère de validation croisée (moyenne des CV_k) :

$$CV = \sum_{k=1}^{K} \frac{n_k}{n} CV_k$$

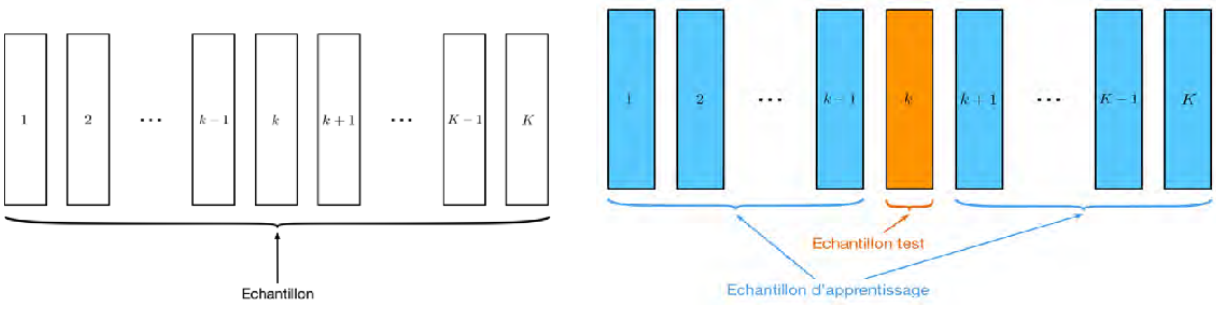

FIGURE 4 – Principe de la validation croisée

Usuellement, $K = 5$ ou $K = 10$. Lorsque $K = n$, on parle d'estimation « *leave one out* ».

À noter que la validation croisée n'est pas la seule technique d'échantillonnage existante : on peut par exemple utiliser du bootstrap, ou une méthode d'estimation *out of bag*.

Démarche globale

Pour synthétiser, la démarche d'un globale d'un projet de statistiques appliquées ou machine learning peut être vue comme suit :

1. Formalisation du problème (avec l'équipe métier !)
2. Recueil et importation des données
3. Nettoyage des données et premières analyses descriptives, visualisations
4. Choix d'une métrique pour évaluer la performance des modèles (avec l'équipe métier !)
5. Découpage de l'échantillon en train/test
6. Estimation des modèles (train) et prédictions (test) $\rightarrow$ mesure des performances
7. Sélection du meilleur modèle

Il n'y a pas un meilleur algorithme qu'une autre, tout dépend de la nature des données, leur volumétrie etc. À noter qu'en cas de performances proches, le choix du modèle se fera souvent sur des critères opérationnels : quel est le modèle le plus interprétable ? le plus simple à comprendre pour la hiérarchie ? le plus simple à mettre en oeuvre ? le plus rapide à calculer ?

Dans ce module nous allons pratiquer d'autres méthodes de machine learning que celles vues jusqu'à présent, en faisant attention à ce compromis biais/variance et en mettant en oeuvre des procédures d'échantillonnage. Nous utiliserons en particulier le package `caret` (Classification and REgression Training) qui permet de calibrer les paramètres de plusieurs centaines d'algorithmes. La documentation complète est disponible ici.

Nous utiliserons le même dataset tout au long du module, `spam`, avec lequel l'objectif sera de prédire si un email est un spam ou un courriel légitime à partir des mots utilisés et la ponctuation. Ce jeu de données se trouve dans le package `kernlab`. Vous pouvez créer le fichier `Module5.R` et charger le jeu de données.

```
library(kernlab)
data(spam)
```

2 Les *k* plus proches voisins (*kNN : k-Nearest Neighbors*)

Le principe est de prédire le label d'une nouvelle observation x à partir des labels des k plus proches voisins de x : si une majorité d'entre-eux sont $= 1$ alors $y = 1$ (spam), si une majorité d'entre-eux sont $= 0$, alors $y = 0$ (mail légitime). À noter que dans le cas d'une variable Y quantitative, on peut faire la moyenne des voisins de x.

L'enjeu est de déterminer k, le nombre de voisins de x à considérer : si k est trop grand on aura toujours tendance à prévoir le label majoritaire dans l'échantillon (biais élevé), si k très petit, très peu d'individus seront utilisés pour la prévision, l'estimateur aura une variance élevée.

Voyons ce que cela donne en utilisant la commande `knn()` du package `class`, avec un exemple avec 3 plus proches voisins. En amont, on construit un échantillon d'entraînement et un échantillon de validation. On évalue la performance de l'algorithme en calculant le taux de courriels mal classés sur l'échantillon de test.

```
library(class)
set.seed(1234) # Pour la reproductibilité

# Echantillonnage
spam1 <- spam[sample(nrow(spam)),]
app <- spam1[1:3000,] # train
valid <- spam1[-(1:3000),] # test

# Algorithme k-nn avec k = 3 plus proches voisins
knnspam <- knn(app[,-58],valid[,-58],cl=app$type,k=3)

# Performance de l'algo : taux de mauvaises prédictions sur valid
mean(knnspam!=valid$type)
```

Avec k=3, on a donc près de 19% de courriels mal classés.
Pour choisir le k le plus pertinent, on pourrait répéter cette méthode pour k=1 à k=100 par exemple, calculer le taux de mauvaises prédictions à chaque fois et choisir k pour lequel l'erreur est minimale.

La fonction `train()` de la librairie `caret` nous permet de faire cela automatiquement. Plusieurs arguments sont à préciser dans cette fonction : l'algorithme, la grille de valeurs de k à tester, le critère dde performance (par défaut : taux de biens classés ou erreur quadratique moyenne) et la méthode d'estimation (validation croisée, bootstrap...).
Avec une estimation par validation simple (dite aussi *hold-out*), répétée plusieurs fois :

```
library(caret)
```

```
# Grille de k
grille.K <- data.frame(k=seq(1,50,by=1))

# Ech test
ctrl1 <- trainControl(method="LGOCV",index=list(1:3000))

# Ech train et estimation
sel.k1 <- train(type~., data=spam1, method="knn",
                trControl=ctrl1,
                tuneGrid=grille.K)

# Performance selon k
sel.k1
plot(sel.k1)

# k avec les meilleures performances
sel.k1$bestTune
```

Les meilleures performances sont obtenues avec $k = 1$, mais $k = 3$ donne un résultat relativement proche.

Et avec une estimation par validation croisée en 5 blocs :

```
# Ech test avec 5 blocs de validation croisée
ctrl2 <- trainControl(method="cv",number=5)

# Ech train et estimation
sel.k2 <- train(type~., data=spam1, method="knn",
                trControl=ctrl2,
                tuneGrid=grille.K)
sel.k2
plot(sel.k2) # le choix de k=1 est plus net

sel.k2$bestTune
```

3 Forêt aléatoire (Random Forest)

Les forêts aléatoires sont une généralisation de la méthode des arbres de décisions vus dans le Module 4. Une forêt est, comme son nom l'indique, un ensemble (une agrégation) d'arbres : ce type d'algorithme fait partie des méthodes de *bagging* qui consistent à agréger des modèles avec un biais faible et une variance forte.
L'aspect *aléatoire* vient de deux raisons :

— La liste des variables servant à chaque étape de construction d'un arbre est choisie aléatoirement
— Les données utilisées pour entraîner l'arbre sont issues d'un tirage aléatoire avec remise (bootstrap)

L'algorithme de construction de chaque arbre est similaire (mais légèrement différent) à l'algorithme CART vu dans le Module 4. Après avoir construit la forêt, la prédiction est simplement une moyenne sur tous les arbres (ou un vote à la majorité lorsqu'il s'agit de classification). Les forêts aléatoires ont souvent de bonnes performances sur les jeux de données complexes, avec beaucoup de variables explicatives. Cette méthode permet de pallier l'instabilité des arbres simples. Son défaut est de perdre la facilité d'interprétation des arbres.

La fonction `randomForest()` du package du même nom permet d'implémenter cet algorithme avec une syntaxe similaire aux fonctions de régression :

```
set.seed(1200)
library(randomForest)

perm <- sample(4601,3000)
app <- spam[perm,]
valid <- spam[-perm,]

foret <- randomForest(type~.,data=app)
foret
```

Il y a plusieurs paramètres optimisables dans la construction d'une forêt aléatoire. Nous allons nous attarder sur deux paramètres :

1. le nombre d'arbres (`ntree`), par défaut = 500 Il est recommandé de le choisir assez élevé. La fonction `plot()` appliquée sur la forêt permet de voir l'évolution du taux d'erreur en fonction du nombre d'arbres (courbe noire : erreur de classification totale, courbe verte : erreur de classification pour les spams, courbe rouge : erreur de classifications pour les

mails légitimes). Dans notre exemple les taux d'erreurs sont stables, on peut donc garder `ntree`= 500 arbres.
À noter que ces erreurs sont calculées sur des données n'ayant pas servi à la construction des arbres en question (erreur out-of-bag) : ce n'est pas utile de refaire une procédure d'échantillonnage pour choisir ce paramètre.

```
plot(foret)
```

2. le nombre de variables choisies aléatoirement pour construire un nouveau noeud (`mtry`), par défaut égal à une fonction du nombre de variables. Pour optimiser ce paramètre, nous allons utiliser la fonction `train()` du package `caret` avec une méthode d'échantillonnage `OOB` = Out Of Bag (bien adaptée aux forêts aléatoires, plus rapide qu'une validation croisée).

```
# Grille de mtry à tester : de 1 à 30
grille.mtry <- data.frame(mtry=seq(1,30,by=1))

# Ech test
ctrlforet <- trainControl(method="oob")

# Ech train et estimation
sel.mtry <- train(type~.,data=app,method="rf",
                  trControl=ctrlforet,
                  tuneGrid=grille.mtry)
sel.mtry
```

Les meilleures performances sont obtenues avec un paramètre `mtry` $= 5$ avec 95.1% de bonnes prédictions (aussi avec `mtry` $= 6$).

4 Régression sous contrainte

Les régressions linéaires simples, multiples et logistiques ont été abordées dans le module 4. Le défaut de ces méthodes est qu'en cas de variables corrélées, ou d'un grand nombre de variables ($> N$ le nombre d'observations), la relation donnant $\hat{\beta}$ n'a pas de sens *(la matrice $X'X$ n'est plus inversible)*. Les régressions sous contrainte (ou régressions *pénalisées*) sont une solution en introduisant un facteur de pénalisation λ qui vient diminuer l'importance des variables

(techniquement, cela permet de rendre la matrice $X'X$ inversible).

Une autre possibilité est de sélectionner les variables à mettre dans le modèle en utilisant un critère d'information qui vient pénaliser les modèles avec beaucoup de variables. Les critères les plus couramment utilisés sont le R^2 ajusté, l'AIC, le BIC, le C_p de Mallows, mais nous ne les détaillerons pas ici.

Une régression pénalisée peut prendre plusieurs formes :

1. la régression **lasso** consiste à **annuler** les coefficients des variables multi corrélées avec le facteur de régularisation (= revient à faire de la sélection de variables)
   $\rightarrow$ ajout du terme $\lambda \sum_j |\beta_j|$ dans la log-vraisemblance à minimiser (mathématiquement, cela revient à un problème de minimisation sous contrainte)
2. la régression **ridge** consiste à **réduire** les coefficients des variables multi corrélées avec le facteur de régularisation
   $\rightarrow$ ajout du terme $\lambda \sum_j \beta_j^2$ dans la log-vraisemblance à minimiser
3. la régression **elasticnet** est une combinaison des deux précédentes, en ajoutant un poids $\alpha \in [0;1]$ qui représente le compromis entre ridge et lasso
   $\rightarrow$ ajout du terme $\lambda[(1-\alpha)\sum_j \beta_j^2 + \alpha \sum_j |\beta_j|]$ dans la log-vraisemblance à minimiser
   $\rightarrow$ si $\alpha = 1$ on est dans le cas d'une régression lasso, si $\alpha = 0$ on est dans le cas d'une régression ridge

À noter que chaque coefficient est pénalisé de la même manière, il faut donc que les variables associés soient du même ordre de grandeur (= réduire les variables au préalable).

Le paramètre λ est à choisir, à optimiser. Quand il est nul, on retrouve les estimateurs classiques des moindres carrés ordinaires. Plus λ augmente, plus la contrainte augmente et les coefficients « rétrécissent ». Dans cas limite où $\lambda = \infty$, les $\hat{\beta}$ sont nuls.
Le package `glmnet` prend comme argument une matrice de variables explicatives et le vecteur de la variable à expliquer ; l'algorithme choisit une grille de λ (on peut la lui spécifier) et retourne des coefficients pour chacune de ces valeurs.

```
library(kernlab)
library(glmnet)
data(spam)
set.seed(5678)

# Découpage train/test :
```

```
perm <- sample(4601,3000)
app <- spam[perm,]
valid <- spam[-perm,]

# Lasso : alpha = 1 (valeur par défaut)
lasso <- glmnet(as.matrix(app[,1:57]),
                            app[,58],
                            family="binomial",
                            alpha=1)

# Ridge : alpha = 0 (valeur par défaut)
ridge <- glmnet(as.matrix(app[,1:57]),
                app[,58],
                family="binomial",
                alpha=0)
```

Affichons les chemins de régularisation de chaque régression, c'est-à-dire la réduction des coefficients en fonction de la valeur de λ :

```
par(mfrow=c(1,2))
plot(lasso, xvar="lambda")
plot(ridge, xvar="lambda")
```

Pour sélectionner λ, le package `glmnet` propose une fonction `cv.glmnet()` effectuant une procédure de validation croisée en 10 blocs. Par exemple pour le lasso :

```
Llasso <- cv.glmnet(as.matrix(app[,1:57]),
                    app[,58],
                    family="binomial",
                    alpha=1)
Llasso$lambda.min # lambda qui min l'erreur de prédiction
Llasso$lambda.1se

plot(Llasso) # évo du critère d'erreur en fonction de lambda
# les lignes verticales correspondent à lambda.min et lambda.1se
```

`lambda.min` correspond au lambda qui minimise l'erreur de prédiction lors de la procédure de validation croisée. `lambda.1se` correspond au λ supérieur à `lambda.min` mais dont l'erreur n'est plus grande que d'un écart-type par rapport à l'erreur de prédiction minimale. En pratique on peut donc choisir n'importe quelle valeur entre `lambda.min` et `lambda.1se`; `lambda.1se` est à privilégier si on souhaite avoir un modèle parcimonieux (choix par défaut dans le cas d'un lasso).

On peut directement faire de la prédiction avec l'objet `cv.glmnet()` que l'on vient de construire et qui prend par défaut `lambda.1se` comme λ :

```
prev.class.lasso <- predict(Llasso,
                            newx=as.matrix(valid[,1:57]),
                            type="class")
table(prev.class.lasso,valid$type)
```

Le package `glmnet` permet aussi de faire des régressions elastic net en ajoutant un paramètre α; cependant `cv.glmnet()` ne permet pas d'optimiser à la fois λ et α par validation croisée simultanément. Pour cela, il faudra utiliser `caret`.

5 Support Vector Machines (SVM)

Les séparateurs à vaste marge (SVM - Support Vector Machines) forment un ensemble d'algorithmes introduits dans le cadre de la classification binaire ($Y = \{0, 1\}$). L'objectif est de trouver un hyperplan dans l'espace des variables explicatives qui sépare au mieux les données. Cette séparation « au mieux » des données, optimale, revient à maximiser la marge, c'est-à-dire la distance qui maximise la distance la plus proche de l'hyperplan.

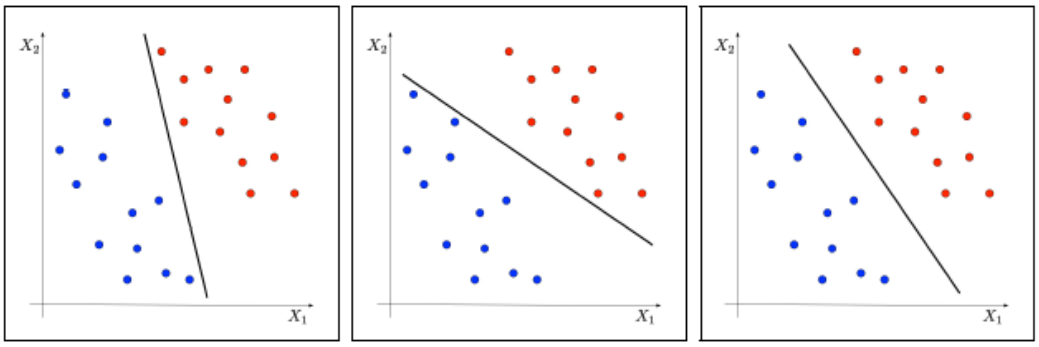

FIGURE 5 – Une infinité d'hyperplans séparateurs possibles

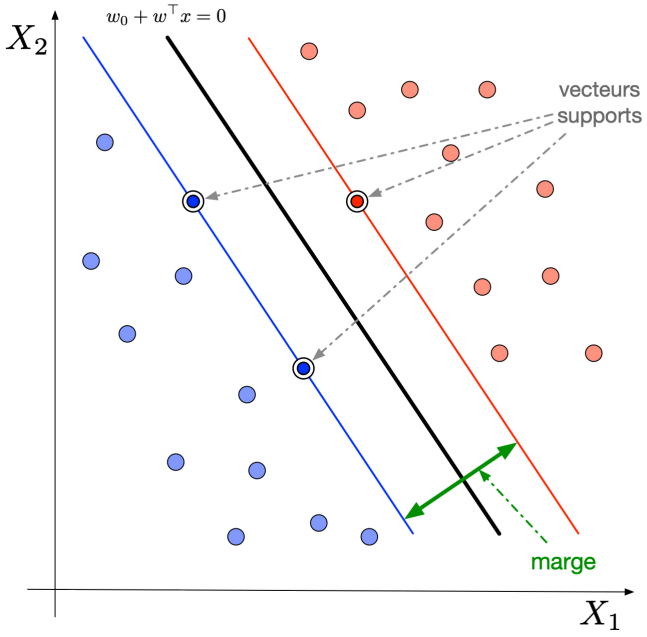

FIGURE 6 – Marge et vecteurs supports

Dans le cas général où les données ne sont pas parfaitement séparables, on autorise des points à se situer dans la marge de l'hyperplan ou à être mal classés (du mauvais côté de l'hyperplan). Un coût est affecté à ces points, appelés variables ressorts. Il y a un compromis à trouver entre le nombre d'erreurs de classification et le niveau de la marge : le paramètre C contrôle ce compromis $\rightarrow$ c'est précisément ce paramètre qui est crucial dans les SVM et qu'on va devoir optimiser.

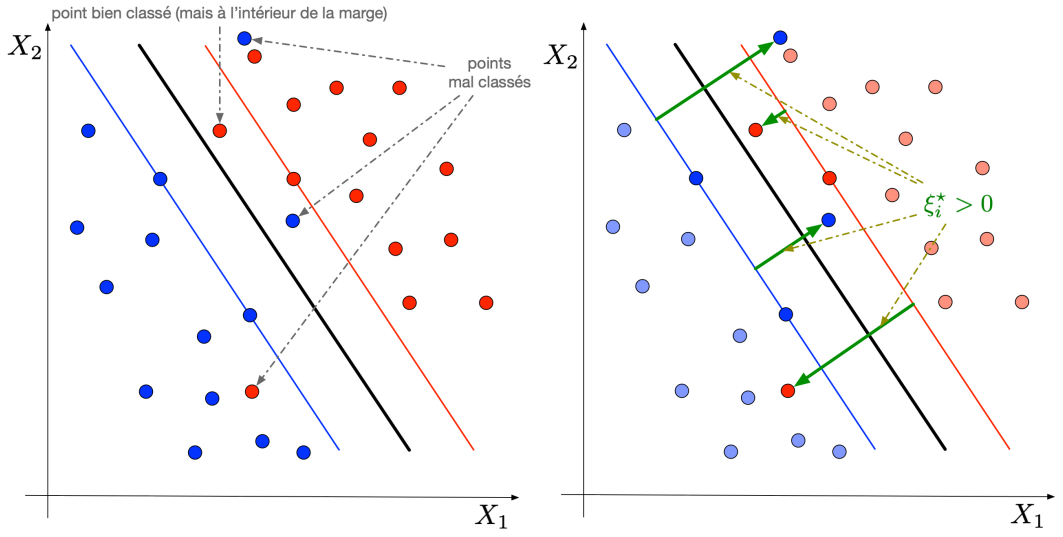

FIGURE 7 – Relâcher les contraintes avec les variables ressorts

Par ailleurs, il est possible d'appliquer une transformation aux données initiales de manière à les rendre linéairement séparable (cela revient à transporter les données dans un nouvel espace, souvent de plus grande dimension) : dans ce cas on applique une SVM en utilisant un noyau (linéaire, polynomial, radial...), d'où le nom d'*astuce du noyau*. Le choix du noyau est un deuxième paramètre à optimiser.

La fonction `ksvm()` du package `kernlab` permet d'implémenter un algorithme SVM :

```
perm <- sample(4601,3000)
app <- spam[perm,]
valid <- spam[-perm,]

# Estimation sur app
mod.svm <- ksvm(type~.,data=app,kernel="vanilladot",C=1)
mod.svm

# Prédictions sur valid
prev.validSVM<-predict(mod.svm,newdata=valid)

table(prev.validSVM,valid$type)
# 62 mails classés à tort comme spams
# 66 mails classés à tort comme non spams
# -> 92% de bonnes prédictions
```

L'option `kernel="vanilladot"` indique qu'on ne transforme pas nos variables (et que nous les transportons pas dans des espaces de plus grandes dimension non plus), *C* est le paramètre de coût.

Dans les SVM, la sélection concommitante de *C* et du noyau est la partie la plus difficile. Une méthode est d'essayer plusieurs noyaux et pour chacun d'eux tester une grille de valeurs pour *C*. La fonction `train()` du package `caret` permet de faire cela avec l'argument `method = "svmPoly"` pour un noyau polynomial et `method = "svmRadial"` pour un noyau radial.

6 Exercice - Module 5

Rappel : les exercices sont à faire en Quarto (ou Rmarkdown) et doivent être rendu dans le même fichier. Vous devez donc compléter votre fichier `Exercice_R_NOM_PRENOM.qmd`.

Pour cet exercice, nous vous laissons choisir le jeu de données de votre choix (jeu de données personnel, données issues de Kaggle, données en *open data*).

Cet exercice doit être vu comme un projet de machine learning : vous devez utiliser différentes méthodes pour faire de la classification/régression (celles vues ici mais vous pouvez également vous renseigner sur d'autres méthodes si cela vous intéresse), les comparer et les commenter avec un regard critique.

Il est indispensable de trouver dans votre projet *a minima* les étapes suivantes :

- Importer et décrire le jeu de données utilisé (donner le lien si disponible en ligne, sinon fournir le jeu de données). En particulier on exposera la variable cible/à prédire et on fera des statistiques descriptives sur les différentes variables explicatives (mesures de tendances centrales et de dispersion, corrélations, graphiques illustrant les liens entre les variables...).
- Énoncer et présenter les différentes techniques envisagées (au moins 3) pour faire de la prédiction.
- Entraîner les modèles choisis en utilisant par exemple le package `caret`. L'objectif est de choisir/optimiser les paramètres adaptés au problème en question. Présenter la ou les stratégie(s) d'optimisation retenues (validation simple, validation croisée...).
- Donner les résultats de vos prédictions sur des échantillons de validation. Comparer les différentes techniques utilisées et conclure.

On attachera de l’importance à commenter chaque étape du projet et à argumenter ses choix. Il ne faut pas se laisser avoir par le côté "presse-bouton" des différents packages et fonctions à la disposition du statisticien et data scientist.