STT3030 Lab 3

2024

Régularisation: Lasso et Ridge

Options par défaut des blocs de code R

Cette option spécifie que le code source R sera affiché dans le document généré.

```
knitr::opts_chunk$set(echo = TRUE)
```

Répertoire de travail

```
getwd()
```

```
## [1] "/Users/agathefernandesmachado/Documents/PhD/STT3030/Labs"
```

```
# A modifier:
#setwd("/Users/agathefernandesmachado/stt3030/lab2")
```

Installation des packages

Une fois installés dans votre environnement, vous n'avez plus besoin d'utiliser ces commandes.

```
#install.packages("ISLR")
#install.packages("nnet")
#install.packages("glmnet")
```

Chargement des packages

```
library(ISLR)
library(nnet)
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

Visualisation du jeu de données

```
head(Hitters)
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun
## -Andy Allanson      293   66     1   30  29    14     1    293    66      1
## -Alan Ashby         315   81     7   24  38    39    14   3449   835     69
## -Alvin Davis        479  130    18   66  72    76     3   1624   457     63
## -Andre Dawson       496  141    20   65  78    37    11   5628  1575    225
## -Andres Galarraga   321   87    10   39  42    30     2    396   101     12
## -Alfredo Griffin    594  169     4   74  51    35    11   4408  1133     19
```

```
##                   CRuns CRBI CWalks League Division PutOuts Assists Errors
## -Andy Allanson       30   29     14      A        E     446      33     20
## -Alan Ashby         321  414    375      N        W     632      43     10
## -Alvin Davis        224  266    263      A        W     880      82     14
## -Andre Dawson       828  838    354      N        E     200      11      3
## -Andres Galarraga    48   46     33      N        E     805      40      4
## -Alfredo Griffin    501  336    194      A        W     282     421     25
##                   Salary NewLeague
## -Andy Allanson        NA         A
## -Alan Ashby        475.0         N
## -Alvin Davis       480.0         A
## -Andre Dawson      500.0         N
## -Andres Galarraga   91.5         N
## -Alfredo Griffin   750.0         A
```

```
?Hitters
```

Quels sont le nombre de lignes et le nombre de colonnes du jeu de données?

```
dim(Hitters)
```

```
## [1] 322  20
```

Y a-t-il des données manquantes? Si oui, combien?

```
cat("Nombre de lignes comportant des données manquantes:", sum(is.na(Hitters)), "\n")
```

```
## Nombre de lignes comportant des données manquantes: 59
```

```
cat("Indice des lignes avec des données manquantes:", which(is.na(Hitters)), "\n")
```

```
## Indice des lignes avec des données manquantes: 5797 5812 5815 5819 5827 5829 5833 5835 5836 5838 583
```

Voici comment supprimer l'ensemble des lignes comportant des données manquantes dans un jeu de données:

```
Hitters <- na.omit(Hitters)
```

Voici les nouvelles dimensions du jeu de données:

```
dim(Hitters)
```

```
## [1] 263  20
```

```
sum(is.na(Hitters)) # on s'assure qu'on a bel et bien retirer les lignes avec valeurs manquantes
```

```
## [1] 0
```

On a donc supprimé:

```
cat(round((322-263)/322*100, 2), "% de lignes", "\n")
```

```
## 18.32 % de lignes
```

On aurait pu remplacer ces données manquantes par des 0 ou la moyenne de la colonne dans le cas d'une variable quantitative, ou également par la médiane de la colonne dans le cas d'une variable qualitative.

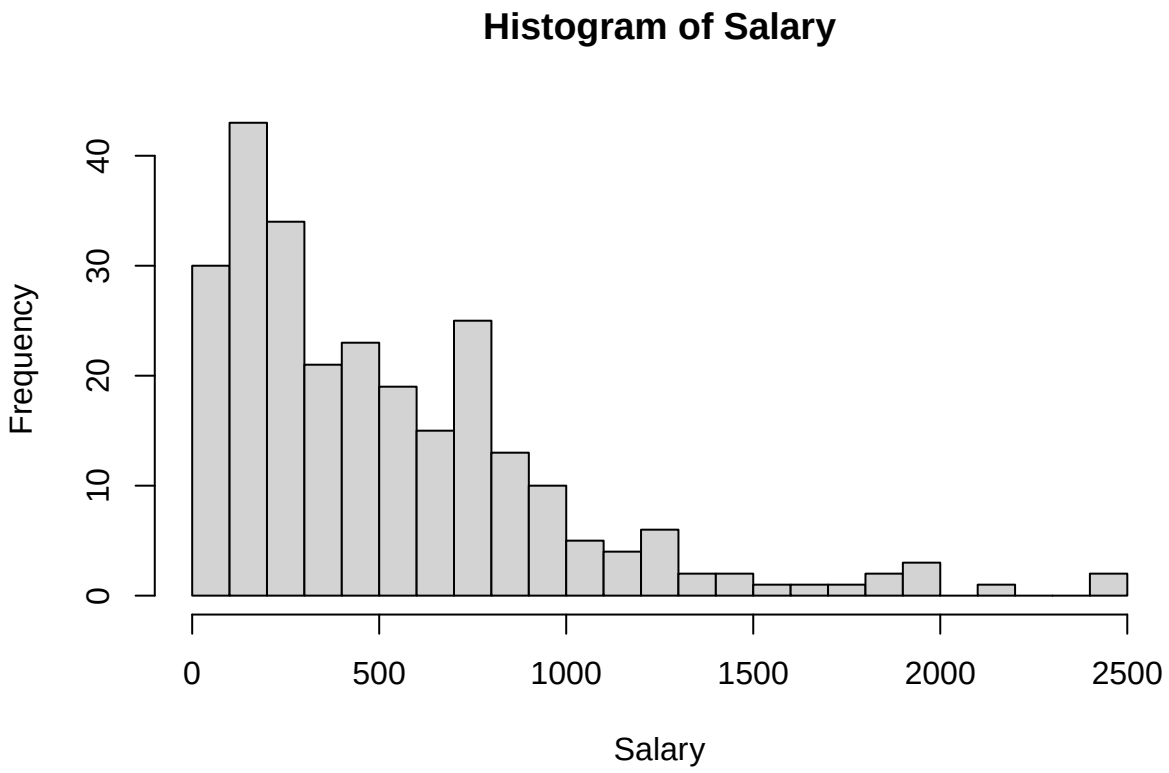
Nom des colonnes du jeu de données:

```
colnames(Hitters)
```

```
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"       "Walks"
##  [7] "Years"     "CAtBat"    "CHits"     "CHmRun"    "CRuns"     "CRBI"
## [13] "CWalks"    "League"    "Division"  "PutOuts"   "Assists"   "Errors"
## [19] "Salary"    "NewLeague"
```

Histogramme de la variable à prédire “Salary”:

```
hist(Hitters$Salary, xlab = "Salary", main = "Histogram of Salary", breaks = 20)
```



Régression Ridge

Nous allons utiliser la fonction “glmnet” (de la librairie R “glmnet”), qui permet de faire de la régression Ridge et Lasso, ainsi qu’un compromis entre ces deux méthodes de régularisation:

```
?glmnet
```

La librairie “glmnet” prend comme argument les variables explicatives sous forme de matrice (“matrix” sous R), contrairement aux fonctions “lm” ou “glm” qui peuvent prendre un objet de type “data.frame” comme entrée.

Le paramètre “alpha” permet de déterminer le type de régularisation qu’on souhaite utiliser: Ridge (“alpha=0”), Lasso (“alpha=1”) ou un compromis entre les deux (https://academic.oup.com/jrsssb/article/67/2/301/7109482).

La variable à prédire y peut être sous forme qualitative ou quantitative, la fonction “glmnet” s’applique en effet aux algorithmes de type Generalized Linear Models (glm).

Il est à noter que la matrice des variables explicatives x, en entrée de la fonction “glmnet”, ne peut contenir que des données numériques.

Quelle est le numéro de colonne de la variable à prédire (“Salary”)?

```
i_Y <- which(colnames(Hitters)=="Salary")
i_Y
```

```
## [1] 19
```

Ainsi, on définit la matrice des variables explicatives du jeu de données “Hitters”:

```
x <- as.matrix(Hitters[,-19])
x[1:3,]
```

```
##                AtBat Hits  HmRun Runs  RBI   Walks Years CAtBat  CHits  CHmRun
## -Alan Ashby   "315" " 81" " 7"  " 24" " 38" " 39" "14"  " 3449" " 835" " 69"
## -Alvin Davis  "479" "130" "18"  " 66" " 72" " 76" " 3"  " 1624" " 457" " 63"
## -Andre Dawson "496" "141" "20"  " 65" " 78" " 37" "11"  " 5628" "1575" "225"
##                CRuns  CRBI   CWalks League Division PutOuts Assists Errors
## -Alan Ashby    " 321" " 414" " 375" "N"    "W"      " 632"  " 43"   "10"
## -Alvin Davis   " 224" " 266" " 263" "A"    "W"      " 880"  " 82"   "14"
## -Andre Dawson " 828" " 838" " 354" "N"    "E"      " 200"  " 11"   " 3"
##                NewLeague
## -Alan Ashby    "N"
## -Alvin Davis   "A"
## -Andre Dawson "N"
```

On remarque plusieurs " " indiquant que les valeurs du jeu de données "Hitters" sont toutes considérées comme étant des chaînes de caractères. On remarqu'il y a à la fois des variables qualitatives ("NewLeague" par exemple) et des variables quantitatives ("Hits" par exemple) dans le jeu de données. Or, la librairie "glmnet" ne prend en entrée que des variables explicatives dont les valeurs sont des données numériques. Il va donc falloir encoder nous-mêmes les chaînes de caractères comme des variables de type numérique, à l'aide de la méthode one-hot encoding (contrairement aux fonctions "lm" et "glm" de R qui gèrent les variables catégorielles/qualitatives en les transformant en variables numériques à l'aide de dummy variables et d'une modalité de référence pour éviter la colinéarité).

Voici une fonction permettant de transformer une matrice comportant des variables quantitatives et qualitatives (toutes sous forme de chaînes de caractères) en données numériques: R opère la technique du one-hot encoding sur les variables quantitatives (R repère lui-même ce type de variables pour les transformer):

```
x <- model.matrix(Salary~., Hitters)[, -1]
x[1:3,]
```

```
##                AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns
## -Alan Ashby      315   81     7   24  38    39    14   3449   835     69   321
## -Alvin Davis     479  130    18   66  72    76     3   1624   457     63   224
## -Andre Dawson    496  141    20   65  78    37    11   5628  1575    225   828
##                CRBI CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## -Alan Ashby     414    375       1         1     632      43     10          1
## -Alvin Davis    266    263       0         1     880      82     14          0
## -Andre Dawson   838    354       1         0     200      11      3          1
```

R, comme dans les fonctions "lm" et "glm", définit une modalité de référence pour les variables catégorielles "League", "Division" et "NewLeague".

On transforme également les observations de la variable à prédire "Salary" en vecteur:

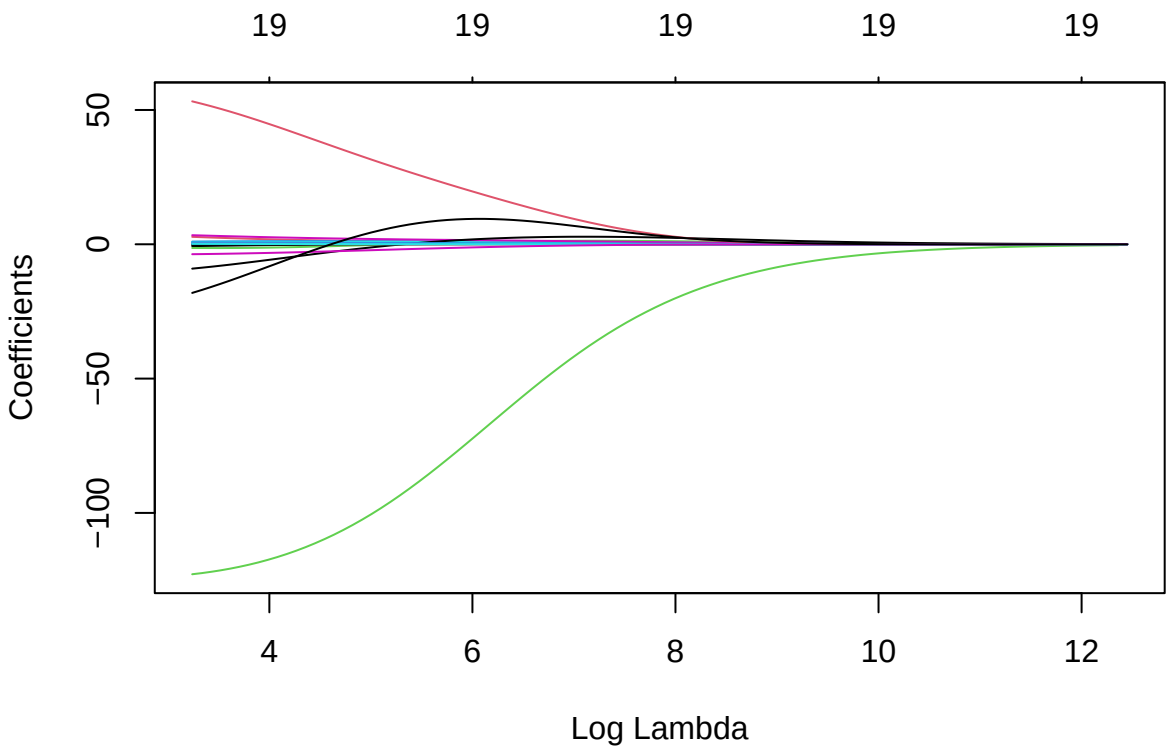
```
y <- Hitters$Salary
```

On peut désormais appliquer la fonction "glmnet" afin d'entraîner un modèle de régression Ridge pour différentes valeurs de λ:

```
ridgefit <- glmnet(x, y, alpha = 0)
```

On peut représenter les différents coefficients du modèle en fonction de la valeur de λ (poids de la pénalité Ridge dans la régression linéaire), en échelle logarithmique:

```
plot(ridgefit, xvar = "lambda")
```




```
dim(coef(ridgefit))
```

```
## [1]  20 100
```

On obtient 20 coefficients (l'intercept ou β_0 et les $\boldsymbol{\beta}$ associés aux variables explicatives du jeu de données "Hitters") estimés pour 100 valeurs différentes de λ.

On peut également obtenir les coefficients estimés à l'aide de la fonction "predict" où "s" indique la valeur du paramètre de pénalité λ:

```
?predict.glmnet
predict(ridgefit , s = 50, type = "coefficients")[1:20,]
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  4.821654e+01 -3.538650e-01  1.953167e+00 -1.285127e+00  1.156329e+00
##           RBI         Walks         Years        CAtBat         CHits
##  8.087771e-01  2.709765e+00 -6.202919e+00  6.085854e-03  1.070832e-01
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##  6.290984e-01  2.172926e-01  2.152888e-01 -1.488961e-01  4.586262e+01
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -1.182304e+02  2.501647e-01  1.208491e-01 -3.277073e+00 -9.423459e+00
```

Maintenant qu'on a compris comment la fonction "glmnet" fonctionne, on va entraîner un modèle de régression Ridge sur le jeu de données "Hitters". On commence donc par séparer la base de données en un échantillon d'entraînement (pour estimer les coefficients du modèle) et un échantillon de test (pour évaluer le modèle obtenu sur de nouvelles données).

```
set.seed(2024)
n <- nrow(x) # Nombre d'observations dans le jeu de données "Hitters"
prop <- 0.8 # Proportion du jeu de données qu'on désire avoir dans l'échantillon d'entraînement (80%)
ntrain <- floor(nrow(x)*prop) # Nombre d'observations souhaité dans l'échantillon d'entraînement (80% *
ntest <- n - ntrain # Nombre d'observations dans l'échantillon de test (20% * n)
```

```
# On mélange les numéros de ligne du jeu de données et on conserve les ntrain premières
train_id <- sample(1:nrow(x), size = ntrain, replace = FALSE)
test <- (-train_id)
```

On mélange les numéros de lignes avant de retenir les ntrain premières observations dans l'échantillon d'entraînement car le jeu de données a pu être ordonné au préalable (selon la variable à prédire "Salary" ou selon tout autre variable).

On retient les observations associées aux numéros de lignes correspondant à ceux tirés pour l'échantillon d'entraînement ("train_id") puis on transforme l'échantillon d'entraînement en objet "matrix" pour pouvoir utiliser la fonction "glmnet".

```
Hitters_train <- Hitters[train_id, ]
x_train <- model.matrix(Salary~., Hitters_train)[,-1]
y_train <- Hitters_train$Salary
dim(x_train) # Dimension de la matrice obtenue
```

```
## [1] 210  19
```

On fait de même pour l'échantillon de test:

```
Hitters_test <- Hitters[(ntrain+1):n,]
x_test <- model.matrix(Salary~., Hitters_test)[,-1]
y_test <- Hitters_test$Salary
dim(x_test)
```

```
## [1] 53 19
```

À partir de maintenant, on met l'échantillon de test de côté, qui ne doit servir que pour l'évaluation du modèle pour ne pas "tricher", et on travaille sur l'échantillon d'entraînement.

On va optimiser la valeur de λ en utilisant la validation croisée. Cela permet de ne pas réserver un échantillon de validation séparé (et donc de perdre des données pour l'entraînement du modèle). Dans le cadre du réglage des hyperparamètres, la validation croisée est employée pour évaluer et sélectionner les meilleures valeurs pour un modèle. Elle implique de diviser les données en k sous-ensembles (ou "folds"). Pour chaque valeur d'hyperparamètre λ, le modèle est entraîné sur une partie des sous-ensembles (généralement sur $k-1$ "folds") et testé sur le sous-ensemble restant. Ensuite, la performance du modèle pour chaque valeur d'hyperparamètre est évaluée sur le sous-ensemble restant. Ce processus est répété k fois, de sorte que chaque "fold" serve à évaluer la performance à un moment donné (comme un échantillon de test). A la fin, les k performances obtenues sont moyennées pour donner une estimation fiable de la performance du modèle pour chaque valeur d'hyperparamètre. La valeur de l'hyperparamètre qui fournit la meilleure performance moyenne est alors choisie. Cette méthode aide à éviter le surapprentissage sur un seul ensemble de données et optimise donc les hyperparamètres de manière plus fiable que si on avait utilisé l'ensemble d'entraînement.

Domaine pour la recherche d'un lambda optimal:

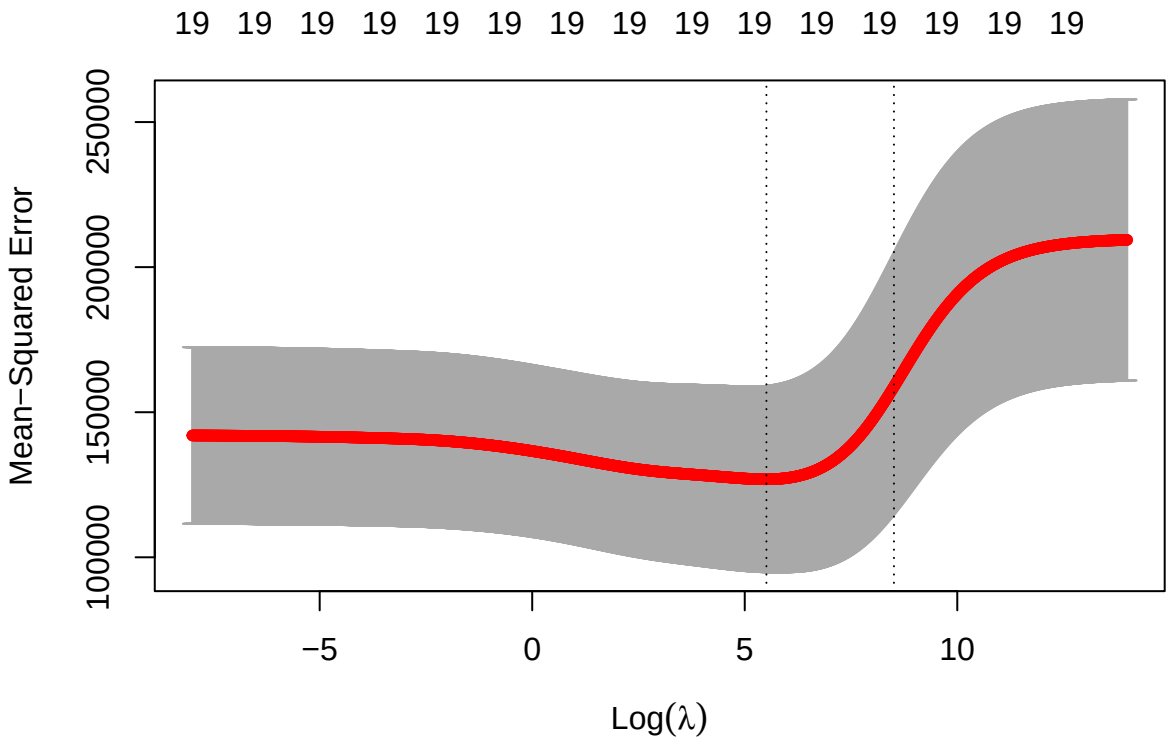
```
tabl <- exp(seq(-8, 14, by = .01))
```

Régression Ridge avec validation croisée:

```
cvfit <- cv.glmnet(x_train, y_train, alpha = 0, lambda = tabl)
# k = nfolds ici (10 par défaut)
```

On peut représenter le graphique donnant la moyenne de l'EQM (sur les k folds) pour les différentes valeurs de λ (contenues dans l'objet "tabl"):

```
plot(cvfit)
```



La première ligne en pointillés correspond à la valeur de λ donnant l'EQM minimale:

```
cvfit$lambda.min # lambda optimal
```

```
## [1] 247.1511
```

```
coef(cvfit, s = "lambda.min")[1:20,] # paramètres optimaux
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  6.109788e+01  3.401553e-03  9.940824e-01 -2.260907e-01  9.804875e-01
##           RBI         Walks         Years        CAtBat         CHits
##  8.338361e-01  1.584234e+00  5.699844e-02  1.328541e-02  6.538970e-02
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##  4.119409e-01  1.170819e-01  1.376029e-01  4.973544e-02  2.785312e+01
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -1.106065e+02  1.859152e-01  8.066905e-02 -2.575482e+00  7.102757e+00
```

```
cvfit$cvm[cvfit$lambda==cvfit$lambda.min] # EQM avec paramètres optimaux
```

```
## [1] 126895.5
```

La deuxième ligne en pointillés correspond à la valeur de λ correspond à la plus grande valeur de λ pour laquelle l'erreur de validation croisée (EQM) est dans une plage d'une erreur standard autour du minimum d'EQM:

```
cvfit$lambda.1se
```

```
## [1] 4964.163
```

```
coef(cvfit, s = "lambda.1se")[1:20,] # paramètres associés à lambda.1se
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
## 321.024558024   0.059274117   0.252289384   0.770564893   0.394315721
##           RBI         Walks         Years        CAtBat         CHits
##   0.400927017   0.498459371   1.996857114   0.005799748   0.021672849
```

```
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##   0.152983641   0.042835574   0.044029835   0.045046708   1.634031046
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -17.172286187   0.032361512   0.005412706  -0.184764346   1.574933875
```

```
cvfit$cvm[cvfit$lambda==cvfit$lambda.1se] # EQM associé à lambda.1se
```

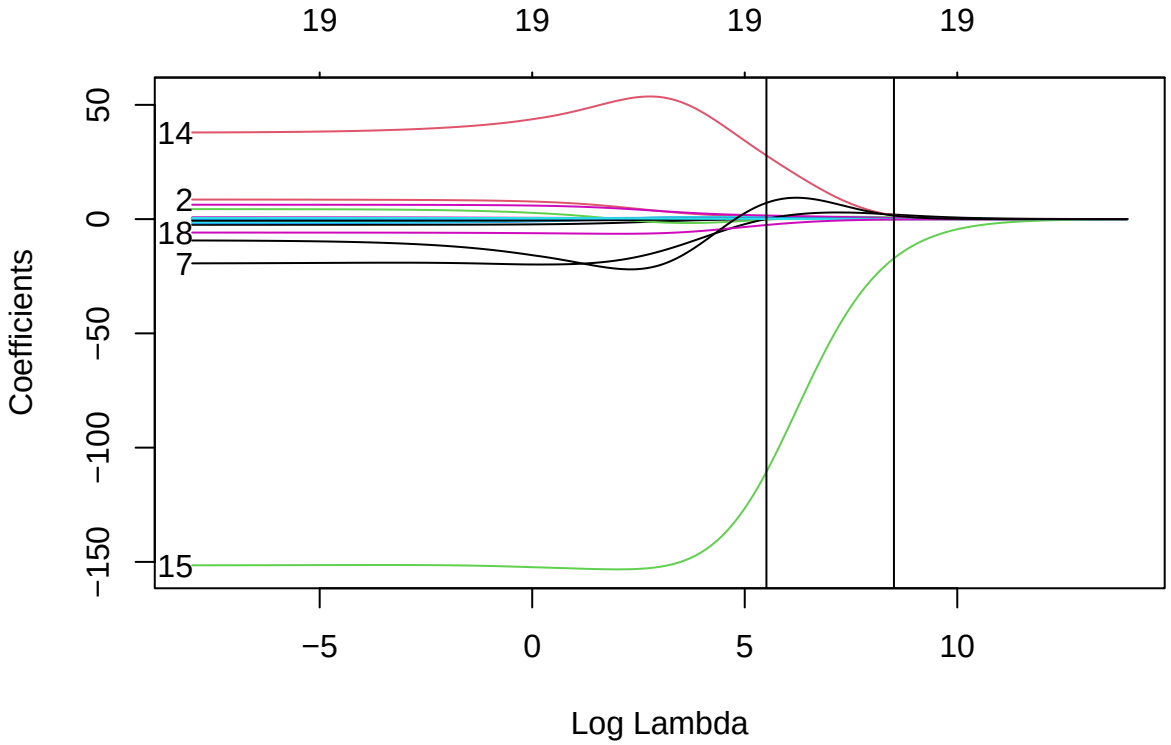
```
## [1] 158795.5
```

L'idée derrière "lambda.1se" est de sélectionner un modèle plus simple, car avec plus de régularisation ($lambda.1se > lambda.min$) conservant une EQM proche de l'EQM minimale.

On connaît désormais la valeur de l'hyperparamètre λ optimale grâce au mécanisme de validation croisée. Observons l'impact sur les coefficients estimés:

```
fit <- glmnet(x_train, y_train, alpha = 0, lambda = tabl)
plot(fit, xvar="lambda")
abline(v = log(cvfit$lambda.min))
abline(v = log(cvfit$lambda.1se))

# On peut ajouter le numéro des coefficients (associé à un numéro de colonnes de x_train) pour chaque c
vnat <- coef(fit)
vnat <- vnat[-1, ncol(vnat)]
axis(2, at = vnat, line = -2, label = as.character(1:19), las = 1, tick = FALSE, cex.axis = 1)
```



On peut prédire le modèle régularisé avec "lambda.min" sur nos données de test.

```
pred <- predict(cvfit, s = cvfit$lambda.min, newx = x_test)
pred[1:5, ]
```

```
##   -Ron Roenicke  -Ryne Sandberg -Rafael Santana      -Rick Schu   -Ruben Sierra
##        438.1527        808.0974        384.5434        263.2168        233.4529
```

On définit une fonction calculant l'EQM afin d'évaluer la performance du modèle. Une bonne pratique de travail est d'ajouter quelques commentaires pour décrire la fonction.

```
# calcul_eqm: retourne l'EQM entre deux vecteurs.
# pred: vecteurs de prédictions
# y: vecteur de vraies valeurs
calcul_EQM <- function(y, preds){
  eqm <- sum((preds - y)^2)/length(y)
  eqm
}
```

On peut calculer l'EQM obtenu sur les données de test avec la régression Ridge:

```
ridge_EQM <- calcul_EQM(y_test, pred)
ridge_EQM
```

```
## [1] 81898.05
```

Régression Lasso

La procédure est exactement la même que pour la régression Ridge sauf qu'on spécifie "alpha = 1" dans la fonction "glmnet".

On garde le même domaine pour la recherche d'un lambda optimal que pour la régression Ridge:

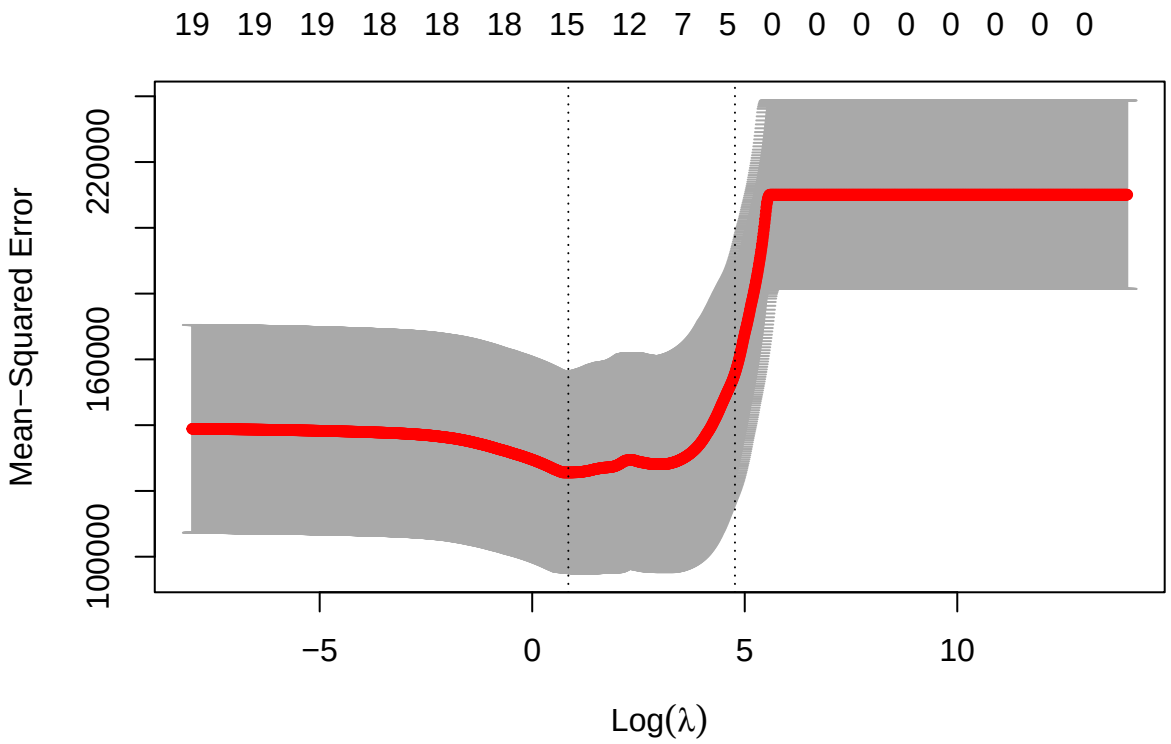
```
tabl <- exp(seq(-8, 14, by = .01))
```

Régression Ridge avec validation croisée:

```
cvfit <- cv.glmnet(x_train, y_train, alpha = 1, lambda = tabl)
# k = nfolds ici (10 par défaut)
```

On peut représenter le graphique donnant la moyenne de l'EQM (sur les k folds) pour les différentes valeurs de λ (contenues dans l'objet "tabl"):

```
plot(cvfit)
```



La première ligne en pointillés correspond à la valeur de λ donnant l'EQM minimale. Afficher la valeur du λ

correspondant, les coefficients ainsi que l'EQM minimale:

```
lambda <- cvfit$lambda.min
cat("Lambda:", lambda, "\n")
```

```
## Lambda: 2.339647
```

```
coef <- coef(cvfit, s = "lambda.min")[1:20,]
cat("Coefficients associés à lambda:", coef, "\n")
```

```
## Coefficients associés à lambda: 228.7611 -1.907014 6.522549 0.0624902 0 0 4.49003 -16.27218 0 0.1436
```

```
eqm <- cvfit$cvm[cvfit$lambda==cvfit$lambda.min]
cat("EQM associée à lambda:", eqm, "\n")
```

```
## EQM associée à lambda: 125618.8
```

Faire de même pour la deuxième ligne en pointillés, correspondant à la plus grande valeur de λ pour laquelle l'erreur de validation croisée (EQM) est dans une plage d'une erreur standard autour du minimum d'EQM:

```
lambda <- cvfit$lambda.1se
cat("Lambda:", lambda, "\n")
```

```
## Lambda: 117.9192
```

```
coef <- coef(cvfit, s = "lambda.1se")[1:20,]
cat("Coefficients associés à lambda:", coef, "\n")
```

```
## Coefficients associés à lambda: 303.0849 0 0.9470735 0 0 0 0.1015129 0 0 0 0 0.060583 0.3373022 0 0 
```

```
eqm <- cvfit$cvm[cvfit$lambda==cvfit$lambda.1se]
cat("EQM associée à lambda:", eqm, "\n")
```

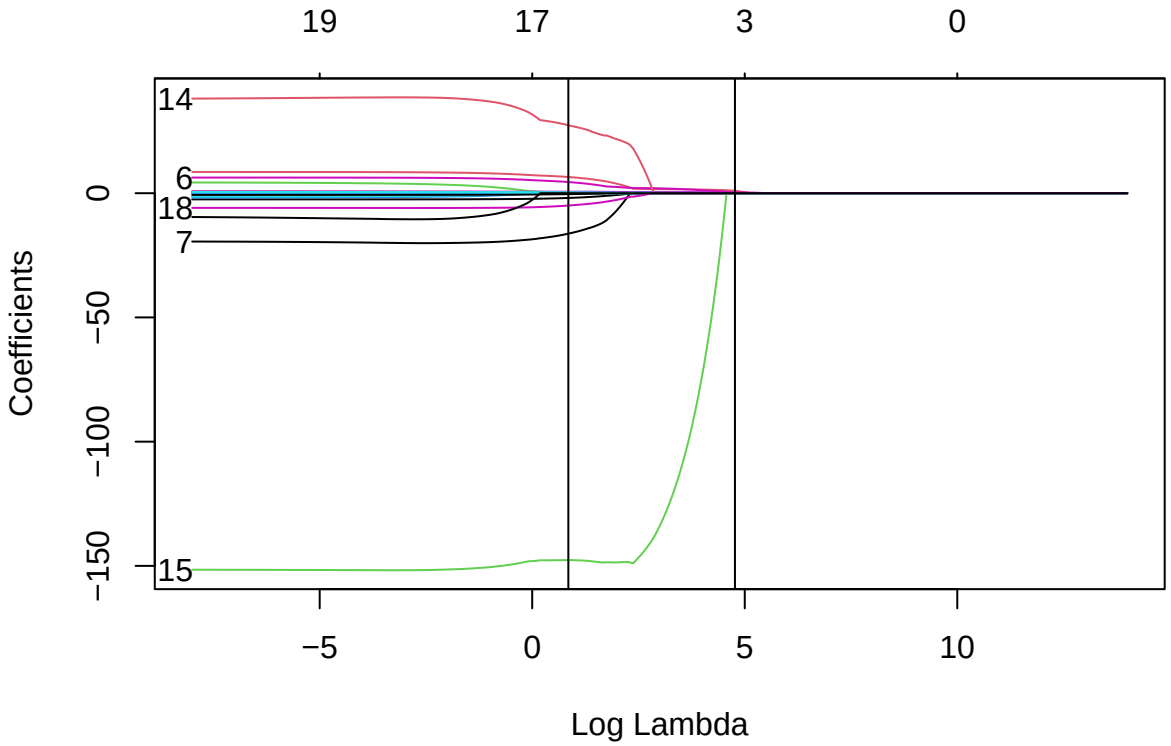
```
## EQM associée à lambda: 155737.4
```

Contrairement à la régression Ridge, on voit clairement le mécanisme de sélection de variables pour Lasso car certains coefficients ont des valeurs égales à 0.

On choisit désormais la valeur de l'hyperparamètre λ optimale ("lambda.min"). Observons l'impact sur les coefficients estimés:

```
fit <- glmnet(x_train, y_train, alpha = 1, lambda = tabl)
plot(fit, xvar="lambda")
abline(v = log(cvfit$lambda.min))
abline(v = log(cvfit$lambda.1se))

# On peut ajouter le numéro des coefficients (associé à un numéro de colonnes de x_train) pour chaque c
vnat <- coef(fit)
vnat <- vnat[-1, ncol(vnat)]
axis(2, at = vnat, line = -2, label = as.character(1:19), las = 1, tick = FALSE, cex.axis = 1)
```



Prédire le modèle régularisé avec Lasso “lambda.min” sur nos données de test et afficher les cinq premières valeurs:

```
pred <- predict(cvfit, s = cvfit$lambda.min, newx = x_test)
pred[1:5, ]
```

```
##   -Ron Roenicke  -Ryne Sandberg -Rafael Santana      -Rick Schu   -Ruben Sierra
##        456.3929        934.7825        342.4133        302.6373        175.0091
```

Calculer l’EQM obtenu sur les données de test avec la régression Lasso, et comparer la valeur avec celle obtenue avec la régression Ridge:

```
lasso_EQM <- calcul_EQM(y_test, pred)
lasso_EQM
```

```
## [1] 81091.38
```

```
ridge_EQM
```

```
## [1] 81898.05
```

Régression linéaire simple

On peut comparer les résultats de régressions Lasso et Ridge avec la régression linéaire simple:

```
lmfit <- lm(Salary~., data = Hitters_train)
pred <- predict(lmfit, newdata = Hitters_test)
lm_EQM <- calcul_EQM(y_test, pred)
lm_EQM
```

```
## [1] 82582.27
```

```
lasso_EQM
```

```
## [1] 81091.38
```

```
ridge_EQM
```

```
## [1] 81898.05
```