



# 1 Introduction

Régressions ridge et elasticnet sous R via les packages « [glmnet](#) » et « [tensorflow / keras](#) ».

Ce tutoriel fait suite au support de cours consacré à la régression régularisée ([RAK, 2018](#)). Il vient en contrepoint au document récent consacré à la [Régression Lasso sous Python](#). Nous travaillons sous R cette fois-ci et nous étudions les régressions ridge et elasticnet.

Nous nous situons dans le cadre de la régression logistique avec une variable cible qualitative binaire. Le contexte n'est pas favorable avec un échantillon d'apprentissage constitué de  $n_{\text{train}} = 200$  observations et  $p = 123$  descripteurs, dont certains sont en réalité des constantes. Les propriétés de régularisation de ridge et elasticnet devraient se révéler décisives. Encore faut-il savoir / pouvoir déterminer les valeurs adéquates des paramètres de ces algorithmes. Ils pèsent fortement sur la qualité des résultats.

Nous verrons comment faire avec les outils à notre disposition. Nous utiliserons les packages "[glmnet](#)" et "[tensorflow / keras](#)". Ce dernier tandem a été présenté plus en détail dans un précédent document ([Avril 2018](#)). Il faut s'y référer notamment pour la partie installation qui n'est pas triviale.

## 2 Données et régression logistique glm()

### 2.1 Base de données « adult »

Nos données sont dérivées de la base « Adult Data Set » accessible sur le serveur UCI (<https://archive.ics.uci.edu/ml/datasets/adult>). Elle a été retravaillée puis publiée sur le site LIBSVM (<https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html> ; base « **a1a** ») : les quantitatives ont été discrétisées, puis transformées en variables indicatrices via un codage disjonctif complet ; les variables catégorielles ont également été binarisées via le même procédé. En définitive, nous disposons de  $p = 123$  variables explicatives, toutes binaires. La variable cible est également binaire, définie dans {positive, negative}.



Nous avons réservé  $n_{\text{train}}=200$  observations (extraits aléatoirement de la base d'apprentissage de « a1a ») pour la modélisation,  $n_{\text{test}} = 30956$  pour l'évaluation.

## 2.2 Importation des bases d'apprentissage et de test – Quelques vérifications

**Echantillon d'apprentissage.** Nous importons l'échantillon d'apprentissage dans un premier temps.

```
#changer Le dossier courant
setwd("... votre dossier ...")

#charger Les données d'apprentissage
DTrain <- read.table("adult_train.txt", sep="\t", header=TRUE)
print(dim(DTrain))

## [1] 200 124
```

Nous vérifions la répartition des classes. Le modèle par défaut consisterait à prédire systématiquement la classe majoritaire "negative".

```
#répartition de La classe
print(prop.table(table(DTrain$classe)))

##
## negative positive
## 0.765 0.235
```

Nous plaçons les variables explicatives et la variable cible dans deux structures distinctes, une matrice pour les premières, un vecteur pour la seconde.

```
#typage
XTrain <- as.matrix(DTrain[,-1])
yTrain <- as.matrix(DTrain[,1])
```

Notons une particularité qui ne va certainement pas faciliter le processus de modélisation : **32 colonnes sont composées de la valeur constante 0.** Logiquement, il faudrait exclure d'emblée ces variables. Elles sont sources de problèmes et ne peuvent en rien contribuer à la qualité prédictive des modèles. Dans notre cas, nous les conservons quand-même pour corser l'affaire et voir justement comment se comporteront les différentes techniques et outils que nous examinerons dans ce tutoriel.



```
#particularité - variables remplies de 0 (somme de la colonne = 0)
print(length(which(colSums(XTrain)==0)))
```

```
## [1] 32
```

**Echantillon test.** Nous chargeons l'échantillon test dans un second temps. Il est composé de `ntest=30956` observations.

```
#charger les données test
DTest <- read.table("adult_test.txt", sep="\t", header=TRUE)
print(dim(DTest))
```

```
## [1] 30956 124
```

```
#répartition des classes
print(prop.table(table(DTest$classe)))
```

```
##
## negative positive
## 0.759465 0.240535
```

La répartition des classes est très similaire à celle de l'échantillon d'apprentissage, heureusement. Avec le modèle par défaut, prédiction systématique de la classe majoritaire "négative", le taux d'erreur serait de 24.05%. Il s'agit de faire mieux avec les différentes variantes de la régression logistique que nous mettrons en œuvre dans ce qui suit.

Nous plaçons également la matrice des descripteurs dans une structure dédiée.

```
#matrice en test
XTest <- as.matrix(DTest[, -1])
```

### 2.3 Régression logistique avec `glm()`

La fonction `glm()` du package « [stats](#) », chargé automatiquement au démarrage, est l'outil privilégié pour la régression logistique sous R. Nous l'appliquons à l'échantillon d'apprentissage. Un message d'avertissement indiquant la non convergence de l'algorithme apparaît. Ce n'est pas bon signe.

```
#rég. logistique usuelle
reg1 <- glm(classe ~ ., data = DTrain, family = "binomial")

## Warning: glm.fit: algorithm did not converge
```



```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Affichons quand-même les résultats par acquit de conscience.

```
print(summary(reg1))
```

```
##
## Call:
## glm(formula = classe ~ ., family = "binomial", data = DTrain)
##
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -1.284e-04 -1.455e-06 -2.100e-08 -2.100e-08  1.274e-04
##
## Coefficients: (51 not defined because of singularities)
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   3119.25 1257717.91   0.002   0.998
## v1             126.84  462342.48   0.000   1.000
## v2             234.25  48861.05   0.005   0.996
## v3            -37.24   9484.46  -0.004   0.997
## v4             194.45  43433.80   0.004   0.996
## v5              NA         NA     NA     NA
## v6           -1048.88 755530.55  -0.001   0.999
## v7            -808.10 207629.46  -0.004   0.997
## v8            -699.60 404974.80  -0.002   0.999
## v9            -876.79 592768.74  -0.001   0.999
## v10           -849.20 697678.04  -0.001   0.999
## v11           -914.76 1955696.16   0.000   1.000
## v12              NA         NA     NA     NA
## v13              NA         NA     NA     NA
## v14           -415.90 298074.01  -0.001   0.999
## v15           -127.64 122628.34  -0.001   0.999
## v16           -394.16  64017.01  -0.006   0.995
## v17           -239.49  68996.62  -0.003   0.997
## v18              NA         NA     NA     NA
## v19           -959.52 603917.71  -0.002   0.999
## v20          -1053.02 674647.75  -0.002   0.999
## v21           -828.53 629664.29  -0.001   0.999
## v22           -997.16 604932.91  -0.002   0.999
## v23           -542.67 1854117.91   0.000   1.000
## v24          -1087.11 648133.88  -0.002   0.999
## v25           -923.57 1459947.72  -0.001   0.999
## v26          -1021.39 943658.84  -0.001   0.999
## v27             568.29 7437369.94   0.000   1.000
## v28           -719.41 723412.79  -0.001   0.999
## v29           -866.74 1709318.04  -0.001   1.000
## v30           -706.65 1655492.49   0.000   1.000
```



## v31	-884.33	624404.14	-0.001	0.999
## v32	-477.04	1067599.14	0.000	1.000
## v33	361.57	2453861.22	0.000	1.000
## v34	NA	NA	NA	NA
## v35	NA	NA	NA	NA
## v36	NA	NA	NA	NA
## v37	NA	NA	NA	NA
## v38	NA	NA	NA	NA
## v39	NA	NA	NA	NA
## v40	-160.58	358503.82	0.000	1.000
## v41	-142.20	833697.03	0.000	1.000
## v42	-600.32	579423.46	-0.001	0.999
## v43	-813.57	767677.93	-0.001	0.999
## v44	-130.51	766046.99	0.000	1.000
## v45	-1336.02	5576038.58	0.000	1.000
## v46	NA	NA	NA	NA
## v47	447.62	1043205.60	0.000	1.000
## v48	571.01	234989.37	0.002	0.998
## v49	638.10	1039703.72	0.001	1.000
## v50	1138.57	164597.44	0.007	0.994
## v51	1049.11	196834.13	0.005	0.996
## v52	934.25	191554.91	0.005	0.996
## v53	1234.65	268241.54	0.005	0.996
## v54	122.51	399646.06	0.000	1.000
## v55	707.68	937172.33	0.001	0.999
## v56	513.39	960884.44	0.001	1.000
## v57	1017.46	161160.01	0.006	0.995
## v58	1267.30	808680.52	0.002	0.999
## v59	NA	NA	NA	NA
## v60	NA	NA	NA	NA
## v61	-358.72	328077.79	-0.001	0.999
## v62	46.15	447845.14	0.000	1.000
## v63	NA	NA	NA	NA
## v64	-127.86	303234.50	0.000	1.000
## v65	16.24	1143209.55	0.000	1.000
## v66	NA	NA	NA	NA
## v67	-185.24	418792.64	0.000	1.000
## v68	-317.80	1197770.84	0.000	1.000
## v69	-368.59	880647.89	0.000	1.000
## v70	-344.99	6688799.47	0.000	1.000
## v71	NA	NA	NA	NA
## v72	160.18	353504.12	0.000	1.000
## v73	NA	NA	NA	NA
## v74	-747.42	102311.29	-0.007	0.994
## v75	NA	NA	NA	NA
## v76	-532.40	674105.32	-0.001	0.999
## v77	NA	NA	NA	NA



## v78	-215.72	1000816.77	0.000	1.000
## v79	330.71	729218.23	0.000	1.000
## v80	129.98	118921.46	0.001	0.999
## v81	493.38	270724.74	0.002	0.999
## v82	NA	NA	NA	NA
## v83	-439.87	485730.40	-0.001	0.999
## v84	NA	NA	NA	NA
## v85	NA	NA	NA	NA
## v86	NA	NA	NA	NA
## v87	NA	NA	NA	NA
## v88	-756.62	648200.81	-0.001	0.999
## v89	NA	NA	NA	NA
## v90	NA	NA	NA	NA
## v91	NA	NA	NA	NA
## v92	NA	NA	NA	NA
## v93	-593.29	1304603.93	0.000	1.000
## v94	90.05	2706022.57	0.000	1.000
## v95	-21.88	1557153.61	0.000	1.000
## v96	NA	NA	NA	NA
## v97	NA	NA	NA	NA
## v98	NA	NA	NA	NA
## v99	NA	NA	NA	NA
## v100	-320.78	2632061.41	0.000	1.000
## v101	NA	NA	NA	NA
## v102	NA	NA	NA	NA
## v103	-1411.04	1670003.99	-0.001	0.999
## v104	NA	NA	NA	NA
## v105	NA	NA	NA	NA
## v106	NA	NA	NA	NA
## v107	-28.54	638726.61	0.000	1.000
## v108	NA	NA	NA	NA
## v109	NA	NA	NA	NA
## v110	NA	NA	NA	NA
## v111	NA	NA	NA	NA
## v112	-1519.31	2506287.81	-0.001	1.000
## v113	NA	NA	NA	NA
## v114	NA	NA	NA	NA
## v115	NA	NA	NA	NA
## v116	NA	NA	NA	NA
## v117	NA	NA	NA	NA
## v118	NA	NA	NA	NA
## v119	73.47	1195355.77	0.000	1.000
## v120	NA	NA	NA	NA
## v121	NA	NA	NA	NA
## v122	NA	NA	NA	NA
## v123	NA	NA	NA	NA
##				



```
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 2.1810e+02 on 199 degrees of freedom
## Residual deviance: 1.4425e-07 on 127 degrees of freedom
## AIC: 146
##
## Number of Fisher Scoring iterations: 25
```

Le modèle est inutilisable. Certains coefficients n'ont pas été estimés et correspondent à la valeur NA (not available). Nous ne pouvons ni les interpréter, ni les déployer sur des individus supplémentaires. De fait, nous n'avons même pas essayé d'appliquer le modèle sur l'échantillon test pour en mesurer les performances.

### 3 La librairie "glmnet"

La librairie "[glmnet](#)" a été développée par des grands noms de la statistique (on a le vertige rien qu'en lisant la liste des personnalités associées au package). Elle est efficace (rapidité des calculs, qualité des résultats), et surtout, elle propose toute une panoplie d'outils qui se révèlent précieux dans l'analyse exploratoire, lors de la recherche des combinaisons adéquates des paramètres  $\lambda$  et  $\alpha$ . Cet aspect est d'autant plus important que, nous ne verrons dans ce tutoriel, le gap de performances entre les modèles selon les valeurs attribuées à ces paramètres peut être important.

Pour rappel, la fonction à optimiser en régression logistique régularisée s'écrit (HASTIE & QIAN, 2016) :

$$J = - \left[ \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} y_i \cdot (\beta_0 + x_i^T \beta) - \log(1 + e^{(\beta_0 + x_i^T \beta)}) \right] + \lambda \left[ \frac{(1 - \alpha)}{2} \|\beta\|_2^2 + \alpha \|\beta\|_1 \right]$$

Où  $\beta_0$  est la constante,  $\beta$  est le vecteur des coefficients appliqués aux variables. On observe que la constante de la régression n'est pas concernée par la régularisation.

$\lambda$  est le coefficient de pénalité ;  $\alpha$  permet d'arbitrer entre la contrainte portant sur les normes L2 (ridge,  $\alpha = 0$ ) et L1 (lasso,  $\alpha = 1$ ) des coefficients.



### 3.1 Régression logistique non-régularisée

Après avoir installé "glmnet" (à faire une seule fois), nous la chargeons (**library**)...

```
#Librairie glmnet  
library(glmnet)  
  
## Loading required package: Matrix  
## Loading required package: foreach  
## Loaded glmnet 2.0-16
```

... nous lançons la régression logistique sans paramètre de régularisation ( $\lambda = 0$ ).

Remarque: Nous désactivons la standardisation (`standardize = FALSE`) des variables explicatives parce que nous savons qu'elles sont toutes binaires, définies de facto sur la même échelle. Sur une base quelconque, en l'absence d'informations précises sur les variables, il est plus prudent d'activer l'option.

```
#régression  
reg2 <- glmnet(XTrain,yTrain,family="binomial",standardize=FALSE,lambda=0)  
print(reg2)  
  
##  
## Call:  glmnet(x = XTrain, y = yTrain, family = "binomial", lambda = 0,  
standardize = FALSE)  
##  
##      Df  %Dev Lambda  
## [1,] 91 0.9999      0  
#affichage des coefficients  
print(reg2$beta)  
## 123 x 1 sparse Matrix of class "dgCMatrix"  
##           s0  
## v1  -3.831772e+01  
## v2   7.189413e+01  
## v3  -3.438143e+01  
## v4   5.627871e+01  
## v5  -2.132226e+01  
## v6  -2.882238e+01  
## v7   3.561587e+01  
## v8   2.724448e+02  
## v9   2.922518e+01  
## v10  4.414104e+01  
## v11 -4.083310e+01  
## v12  .  
## v13  .
```





```
## v14 -8.435174e+01
## v15 1.775369e+01
## v16 -8.996599e+01
## v17 -2.786420e+01
## v18 5.995881e+01
## v19 1.632340e+01
## v20 -2.246301e+01
## v21 9.442261e+00
## v22 1.741634e+00
## v23 2.511040e+02
## v24 -3.502176e+01
## v25 3.719931e+01
## v26 -1.571951e+02
## v27 -3.284440e+03
## v28 1.250029e+02
## v29 1.006888e+02
## v30 1.156274e+02
## v31 4.077371e+01
## v32 2.041364e+02
## v33 2.031696e+02
## v34 2.615885e+02
## v35 2.083352e+01
## v36 -3.172136e+00
## v37 9.642887e+00
## v38 5.115858e+00
## v39 -3.752186e+00
## v40 1.673466e+02
## v41 1.079230e+02
## v42 -5.072438e+01
## v43 -1.214190e+02
## v44 9.943991e+01
## v45 2.447386e+03
## v46 2.134736e+03
## v47 -2.811181e+02
## v48 -1.521966e+02
## v49 -1.521847e+02
## v50 5.649339e+01
## v51 2.915158e+01
## v52 -1.493596e+01
## v53 8.305544e+01
## v54 -3.232951e+02
## v55 -8.721700e+01
## v56 -1.419358e+02
## v57 1.015704e+01
## v58 5.422032e+01
## v59 -3.907864e+02
## v60 .
```



```
## v61 -1.599100e+02
## v62 6.274852e+01
## v63 -4.930287e+00
## v64 -1.356651e+01
## v65 8.721319e+01
## v66 7.759744e+01
## v67 -1.100185e+02
## v68 -1.951398e+02
## v69 -2.100170e+02
## v70 3.631192e+03
## v71 8.941748e-01
## v72 7.998418e+01
## v73 1.969641e-13
## v74 -2.938848e+02
## v75 4.264162e-12
## v76 -2.391689e+02
## v77 1.032831e-11
## v78 -1.031697e+02
## v79 1.100999e+02
## v80 2.100086e+01
## v81 1.540110e+02
## v82 -2.227176e+01
## v83 -1.287332e+02
## v84 .
## v85 .
## v86 .
## v87 .
## v88 -2.330019e+02
## v89 .
## v90 .
## v91 .
## v92 .
## v93 -1.292385e+02
## v94 1.310000e+02
## v95 6.400213e-01
## v96 .
## v97 .
## v98 -2.451660e+03
## v99 3.986162e+03
## v100 -5.796967e+01
## v101 .
## v102 .
## v103 -4.517764e+02
## v104 .
## v105 .
## v106 .
## v107 1.130853e+02
```



```
## v108 .
## v109 .
## v110 .
## v111 .
## v112 -2.398007e+02
## v113 .
## v114 .
## v115 .
## v116 .
## v117 .
## v118 .
## v119 1.253353e+02
## v120 .
## v121 .
## v122 .
## v123 .
```

Très curieusement, de par l'algorithme d'optimisation utilisé, le processus semble converger malgré tout contrairement à `glm()` de « stats ». La valeur zéro est attribuée aux coefficients associés aux colonnes de constante. Ces variables sont inactives pour la prédiction.

Nous appliquons le modèle sur l'échantillon test avec **`predict()`**.

```
#prédiction
yp2 <- predict(reg2,XTest,type="class",s=c(0))
print(table(yp2))
```

```
## yp2
## negative positive
## 19568 11388
```

Les prédictions sont réparties entre les deux classes, il n'y a pas de prédiction systématique...

```
#taux d'erreur
print(sum(DTest$classe != yp2)/nrow(DTest))
```

```
## [1] 0.3150924
```

... mais le modèle fait pire que la prédiction par défaut avec un taux d'erreur de 31.5%.

Conclusion: Manifestement, les difficultés (le ratio  $p/n_{\text{train}}$  élevé et, surtout, les variables constituées de constantes) ont eu raison de l'algorithme d'apprentissage en l'absence de contraintes sur les coefficients. Cela montre combien l'inspection des variables préalablement



à l'apprentissage est très importante. Il aurait fallu exclure d'emblée ces variables à problème. Nous continuons en l'état néanmoins en espérant que les mécanismes de régularisation des régressions ridge et elasticnet nous sortiront d'affaire.

### 3.2 Régression Ridge

**Paramétrage et résultats de la régression.** Pour `glmnet()`, la régression ridge correspond à ( $\lambda > 0$ ) et ( $\alpha = 0$ ). Lors de l'appel de la fonction, nous indiquons bien (`alpha = 0`) et nous laissons l'outil déterminer une séquence de (`nlambda = 100`, paramétrable) valeurs de  $\lambda_i$  à explorer. Le mécanisme de détermination des extremums de la plage ( $\lambda_{\max}$ ,  $\lambda_{\min}$ ) est décrit dans un des articles fondateurs des auteurs du package (FRIEDMAN et al., 2010 ; section 2.5 pour la régression<sup>1</sup>). L'autre approche consiste à fixer nous-même la liste des valeurs de  $\lambda$  à tester, soit parce que nous en avons une idée précise (ça reste difficile quand même), soit parce que nous souhaitons comparer des solutions alternatives pour différentes valeurs de  $\alpha$ .

```
#Régression Ridge  
ridge2 <- glmnet(XTrain,yTrain,family="binomial",standardize=FALSE,alpha=0)  
plot(ridge2,xvar="lambda")
```

A l'issue de l'apprentissage, nous disposons d'un vecteur de coefficients  $\beta^i$  pour chaque  $\lambda_i$ . Plus  $\lambda_i$  augmente, plus la norme de  $\beta^i$  diminue. Tous les coefficients sont nuls lorsque  $\lambda = \lambda_{\max}$ . Nous pouvons afficher une « Ridge path coefficients » permettant de juger de la dispersion des coefficients au regard de  $\lambda$  (Figure 1). Rappelons qu'à la différence de la régression Lasso, Ridge ne sait pas fixer sélectivement les coefficients à la valeur 0 et ne permet donc pas de réaliser une sélection de variables.

---

<sup>1</sup> Les corrélations avec la cible, le ratio nombre de variables / nombre d'observations, la valeur de  $\alpha$ , la taille de l'échantillon, le nombre de valeurs à explorer... entrent en jeu.

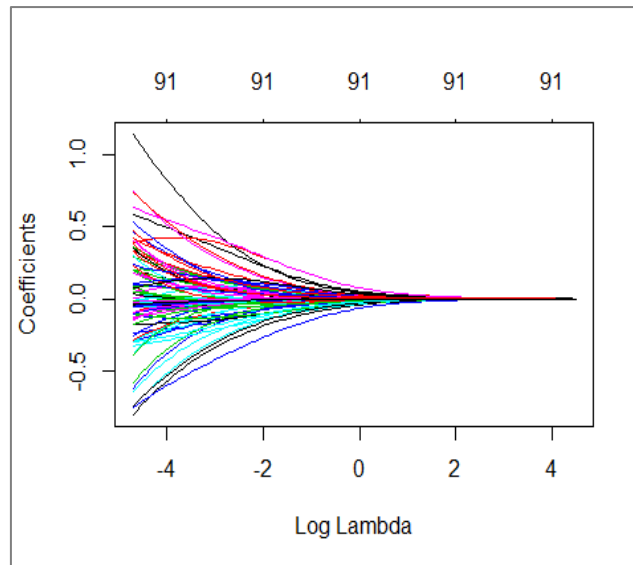


Figure 1 - Ridge path coefficients

**Identification de la valeur « optimale » de  $\lambda$ .** Nous avons les logarithmes des  $\lambda_i$  en abscisse de notre graphique. Ils varient de  $\log(0.008982)=-4.7124$  à  $\log(89.825)=4.4978$ . Une règle empirique consiste à choisir la valeur de  $\lambda$  à partir de laquelle les coefficients commencent « à se stabiliser ». On se rend compte que la lecture en ce sens du graphique n'est pas aisée (Figure 1). De plus, cette démarche ne nous assure pas de trouver la solution qui optimise les qualités prédictives du modèle.

On lui préfère une démarche plus pragmatique visant à optimiser explicitement une mesure d'évaluation des performances. Notre échantillon test devant jouer le rôle de juge impartial, il est hors de question qu'il intervienne dans ce processus. Et, malheureusement, la taille de notre échantillon d'apprentissage est trop faible ( $n_{\text{train}} = 200$ ) pour que nous puissions le scinder en deux parties encore. La solution viable consiste à passer par la validation croisée, qui ne fait intervenir que l'échantillon d'apprentissage. Nous faisons appel à la fonction **cv.glmnet()**

```
#Optimisation du paramètre Lambda  
set.seed(1)  
cv.ridge2 <- cv.glmnet(XTrain,yTrain,family="binomial",type.measure="class",  
                      nfolds=10,alpha=0,keep=TRUE)
```



Par rapport à `glmnet()`, de nouveaux paramètres apparaissent :

- (`type.measure="class"`) indique que nous traitons d'un problème de classement et que le critère utilisé sera le taux d'erreur (*misclassification error*).
- (`nfolds = 10`) pour indiquer une validation croisée en 10 blocs (folds).
- (`keep = TRUE`) pour que l'on conserve en mémoire les groupes de la validation croisée afin de pouvoir reconduire à l'identique l'expérimentation si d'aventure nous souhaitons la relancer avec d'autres jeux de paramètres. Nous reviendrons sur cette question ci-dessous.

Un graphique permet de mettre en relation les valeurs de  $\log(\lambda)$  avec le taux d'erreur moyen en validation croisée (Figure 2, points rouges). Un intervalle de confiance est proposé, défini par  $\pm 1$  écart-type de l'erreur en validation croisée.

```
plot(cv.ridge2)
```

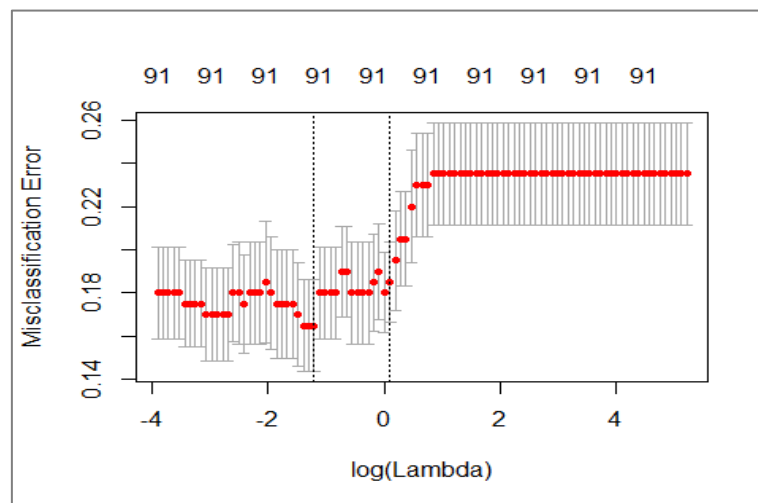


Figure 2 – Ridge - Taux d'erreur en validation croisée vs.  $\log(\lambda)$

**Remarque :** Subdivision en blocs des observations. La validation croisée subdivise les données en groupes pour répéter les processus d'apprentissage et test. L'option "`keep`" permet d'identifier les blocs d'appartenance de chaque individu et pouvoir ainsi répéter à l'identique la démarche pour d'autres modèles. Cette forme d'appariement rend plus puissant les comparaisons des performances, ce dont on ne se privera pas dans la suite de notre étude ci-



dessous. Les indicatrices de blocs sont conservées dans le champ `$foldid`. Nous voyons ainsi que le premier individu a été affecté au bloc n°4, le second au n°5, le troisième au n°4, etc.

```
#subdivision en folds (blocs) des données
```

```
print(cv.ridge2$foldid)
```

```
## [1] 4 5 4 9 10 6 4 8 1 2 6 4 10 2 4 3 3 2 10 1 9 8 6
## [24] 3 8 8 3 7 10 9 2 2 3 2 8 1 1 8 8 3 2 3 4 7 8 3
## [47] 4 4 2 5 7 9 5 6 1 5 6 9 5 8 3 1 4 6 9 5 8 10 1
## [70] 5 5 9 8 3 1 2 8 5 2 7 3 5 5 9 8 4 10 4 8 6 7 7
## [93] 3 4 6 4 2 7 6 2 6 5 10 7 6 1 3 10 6 5 8 10 7 7 7
## [116] 2 6 9 7 2 10 10 7 3 1 9 8 5 7 9 9 6 4 4 1 2 7 9
## [139] 6 1 8 6 7 5 10 5 10 9 9 5 1 5 3 2 4 4 7 4 10 1 2
## [162] 3 2 9 10 7 5 10 1 6 9 1 1 10 10 7 4 3 6 10 1 1 9 6
## [185] 4 3 8 6 3 9 8 2 1 7 9 8 2 5 3 10
```

Revenons aux résultats de la validation croisée. Nous avons accès aux détails avec les propriétés de l'objet généré par la fonction. Nous affichons ci-dessous la suite de  $\lambda_i$  ; le taux d'erreur associé ; le nombre de coefficients non-nuls (qui n'évolue pas puisque nous utilisons une régression ridge).

```
#affichage
```

```
print(cbind(cv.ridge2$lambda,cv.ridge2$cvm,cv.ridge2$nzzero))
```

```
##          [,1] [,2] [,3]
## s0  183.97398884 0.235  91
## s1  167.63024674 0.235  91
## s2  152.73843764 0.235  91
## s3  139.16957582 0.235  91
## s4  126.80613428 0.235  91
## s5  115.54102682 0.235  91
## s6  105.27668045 0.235  91
## s7   95.92419032 0.235  91
## s8   87.40254964 0.235  91
## s9   79.63794803 0.235  91
## s10  72.56313222 0.235  91
## s11  66.11682355 0.235  91
## s12  60.24318718 0.235  91
## s13  54.89134847 0.235  91
## s14  50.01495236 0.235  91
## s15  45.57176185 0.235  91
## s16  41.52329213 0.235  91
## s17  37.83447730 0.235  91
## s18  34.47336662 0.235  91
## s19  31.41084774 0.235  91
```



```
## s20 28.62039460 0.235 91
## s21 26.07783763 0.235 91
## s22 23.76115441 0.235 91
## s23 21.65027894 0.235 91
## s24 19.72692784 0.235 91
## s25 17.97444194 0.235 91
## s26 16.37764206 0.235 91
## s27 14.92269748 0.235 91
## s28 13.59700616 0.235 91
## s29 12.38908561 0.235 91
## s30 11.28847339 0.235 91
## s31 10.28563653 0.235 91
## s32 9.37188893 0.235 91
## s33 8.53931615 0.235 91
## s34 7.78070684 0.235 91
## s35 7.08949029 0.235 91
## s36 6.45967951 0.235 91
## s37 5.88581939 0.235 91
## s38 5.36293941 0.235 91
## s39 4.88651066 0.235 91
## s40 4.45240652 0.235 91
## s41 4.05686700 0.235 91
## s42 3.69646611 0.235 91
## s43 3.36808225 0.235 91
## s44 3.06887111 0.235 91
## s45 2.79624106 0.235 91
## s46 2.54783071 0.235 91
## s47 2.32148845 0.235 91
## s48 2.11525382 0.230 91
## s49 1.92734050 0.230 91
## s50 1.75612088 0.230 91
## s51 1.60011194 0.220 91
## s52 1.45796241 0.205 91
## s53 1.32844104 0.205 91
## s54 1.21042600 0.195 91
## s55 1.10289509 0.185 91
## s56 1.00491693 0.180 91
## s57 0.91564288 0.190 91
## s58 0.83429969 0.185 91
## s59 0.76018281 0.180 91
## s60 0.69265026 0.180 91
## s61 0.63111712 0.180 91
## s62 0.57505042 0.180 91
## s63 0.52396452 0.190 91
## s64 0.47741696 0.190 91
## s65 0.43500455 0.180 91
## s66 0.39635995 0.180 91
```





```
## s67 0.36114842 0.180 91
## s68 0.32906499 0.180 91
## s69 0.29983176 0.165 91
## s70 0.27319554 0.165 91
## s71 0.24892560 0.165 91
## s72 0.22681173 0.170 91
## s73 0.20666240 0.175 91
## s74 0.18830309 0.175 91
## s75 0.17157476 0.175 91
## s76 0.15633253 0.175 91
## s77 0.14244438 0.180 91
## s78 0.12979001 0.185 91
## s79 0.11825983 0.180 91
## s80 0.10775395 0.180 91
## s81 0.09818139 0.180 91
## s82 0.08945922 0.175 91
## s83 0.08151191 0.180 91
## s84 0.07427062 0.180 91
## s85 0.06767262 0.170 91
## s86 0.06166077 0.170 91
## s87 0.05618300 0.170 91
## s88 0.05119186 0.170 91
## s89 0.04664411 0.170 91
## s90 0.04250038 0.175 91
## s91 0.03872476 0.175 91
## s92 0.03528456 0.175 91
## s93 0.03214998 0.175 91
## s94 0.02929386 0.180 91
## s95 0.02669148 0.180 91
## s96 0.02432028 0.180 91
## s97 0.02215973 0.180 91
## s98 0.02019112 0.180 91
```

Identifier visuellement les éléments importants dans cette liste n'est pas aisé. Nous le faisons par le calcul. Tout d'abord, nous affichons l'erreur minimale.

```
#min de l'erreur en cross-validation
```

```
print(min(cv.ridge2$cvm))
```

```
## [1] 0.165
```

Puis nous cherchons le  $\lambda^*$  correspondant à cette erreur.

```
#lambda qui minimise l'erreur
```

```
print(cv.ridge2$lambda.min)
```

```
## [1] 0.2998318
```



Et nous calculons son logarithme.

```
#son Logarithme  
print(log(cv.ridge2$lambda.min))  
## [1] -1.204534
```

Cette coordonnée est matérialisée par le premier trait pointillé (à gauche) dans le graphique de la validation croisée (Figure 2).

**Règle de l'écart-type.** On perçoit un second trait dans la Figure 2 (à droite). Il caractérise la plus grande valeur  $\lambda^{**}$  de  $\lambda$  telle que son erreur moyenne en validation croisée (point rouge) est inférieure à la borne haute de l'intervalle de confiance de l'erreur optimale (pour  $\lambda^*$ ). Quel est son intérêt ? Un peu comme le principe de parcimonie et la préférence à la simplicité des modèles, cette solution témoigne d'une préférence à la régularisation : « à performances similaires sur notre échantillon d'apprentissage, on préfère la solution la plus fortement régularisée, la moins dépendante des données d'apprentissage ». Remarque : Cette « règle de l'écart-type » est un héritage de la méthode d'induction d'arbres CART (Breiman et al., 1984) où l'on essaie de déterminer le plus petit arbre avec un niveau de performances satisfaisant.

Nous accédons à ce  $\lambda^{**}$  et à son logarithme avec les instructions suivantes.

```
#lambda le plus élevé en 1-se rule  
print(cv.ridge2$lambda.1se)  
## [1] 1.102895  
  
#son Log  
print(log(cv.ridge2$lambda.1se))  
## [1] 0.09793863
```

**Inspection des coefficients  $\beta$  pour  $\lambda = \lambda^*$ .** Nous avons accès au vecteur de coefficients de la régression pour  $\lambda$  fixé. Nous affichons ci-dessous les coefficients  $\beta^*$  pour  $\lambda = \lambda^*$ .

```
#coefficients de la régression pour le min  
print(coef(cv.ridge2, s="lambda.min"))  
## 124 x 1 sparse Matrix of class "dgCMatrix"  
## 1
```



```
## (Intercept) -0.782523225
## v1          -0.322461564
## v2          -0.037252283
## v3           0.150885154
## v4           0.224801754
## v5           0.020028216
## v6          -0.012711146
## v7           0.105246716
## v8           0.645204345
## v9           0.144448780
## v10          0.012049813
## v11          0.020331488
## v12          .
## v13          .
## v14          0.015741025
## v15          -0.032479619
## v16          -0.026459669
## v17          0.134858952
## v18          -0.074524174
## v19          0.023037455
## v20          -0.123378657
## v21          -0.073003805
## v22          -0.092995057
## v23          0.692019337
## v24          -0.166082223
## v25          0.127429003
## v26          -0.145467596
## v27          0.509870285
## v28          -0.170048013
## v29          0.419374709
## v30          -0.316873815
## v31          0.080296856
## v32          1.155227185
## v33          -0.186956990
## v34          -0.139242185
## v35          -0.052899328
## v36          -0.092953390
## v37          -0.123367140
## v38          -0.020179377
## v39          0.257540952
## v40          0.321185785
## v41          -0.064749210
## v42          -0.311278670
## v43          -0.331143208
## v44          0.299139978
## v45          0.551363701
## v46          1.339145437
```



```
## v47      0.189309529
## v48     -0.166449634
## v49     -0.335850372
## v50      0.172968308
## v51      0.609440674
## v52      0.197969577
## v53      0.077910478
## v54     -0.434944230
## v55     -0.009649491
## v56      0.097036344
## v57      0.007333934
## v58     -0.194011908
## v59     -0.563868686
## v60      .
## v61      0.096160851
## v62     -0.226794953
## v63      0.350574248
## v64     -0.176103636
## v65     -0.295812477
## v66     -0.022868046
## v67      0.123518325
## v68     -0.302654470
## v69     -0.397195536
## v70     -0.312648819
## v71      0.004614765
## v72     -0.099504384
## v73      0.099498660
## v74     -0.586185604
## v75      0.586107661
## v76     -0.608700944
## v77      0.608591700
## v78     -0.114709087
## v79      0.307196706
## v80     -0.103571495
## v81     -0.048817928
## v82      0.152205727
## v83      0.215262406
## v84      .
## v85      .
## v86      .
## v87      .
## v88     -1.117528181
## v89      .
## v90      .
## v91      .
## v92      .
## v93     -0.489421968
```



```
## v94      -0.242397976
## v95      -0.446869050
## v96      .
## v97      .
## v98      -0.230220549
## v99      1.327426606
## v100     -0.206408979
## v101     .
## v102     .
## v103     -0.200855666
## v104     .
## v105     .
## v106     .
## v107     -0.185250959
## v108     .
## v109     .
## v110     .
## v111     .
## v112     -0.184139264
## v113     .
## v114     .
## v115     .
## v116     .
## v117     .
## v118     .
## v119     -0.196499488
## v120     .
## v121     .
## v122     .
## v123     .
```

Les coefficients nuls (.) correspondent aux variables composées de 0.

Remarque : Nous ne le faisons pas ici, mais, les variables explicatives étant exprimées dans les mêmes unités, les trier selon la valeur absolue décroissante des coefficients permettrait de les hiérarchiser et distinguer celles qui sont les plus influentes dans la régression.

**Prédiction sur l'échantillon test.** Nous pouvons produire plusieurs prédictions à partir des jeux de coefficients  $\beta$  correspondant à différentes valeurs de  $\lambda$ . Ci-dessous, nous appliquons les coefficients  $\beta^*(\lambda^*)$  et  $\beta^{**}(\lambda^{**})$ .

*#prédiction*

```
yr2 <- predict(cv.ridge2,XTest,s=c(cv.ridge2$lambda.min,cv.ridge2$lambda.1se),type="class")
```



Nous avons une matrice avec autant de colonnes que de valeurs de  $\lambda$  essayé. Le nombre de lignes correspond à  $n_{\text{test}}$ , l'effectif de l'échantillon test. Voici les 6 premières lignes.

```
print(head(yr2))  
  
##      1      2  
## [1,] "positive" "negative"  
## [2,] "negative" "negative"  
## [3,] "negative" "negative"  
## [4,] "negative" "negative"  
## [5,] "negative" "negative"  
## [6,] "negative" "negative"
```

Nous calculons les taux d'erreur pour les deux prédictions.

```
#erreur : Lambda.min  
print(sum(DTest$classe != yr2[,1])/nrow(DTest))  
  
## [1] 0.1769932  
#erreur : Lambda.1se  
print(sum(DTest$classe != yr2[,2])/nrow(DTest))  
  
## [1] 0.2110738
```

Les deux font mieux que la prédiction systématique. Malgré les embuches, la régression ridge a su tirer parti des données. On note également que le modèle avec  $\lambda^*$  est meilleur (17.69%). La préférence à la régularisation n'est pas payante dans notre configuration.

### 3.3 Régression elasticnet

**Régression elasticnet – Etude des coefficients.** Pour la régression elasticnet, nous devons paramétrer ( $\lambda > 0$  et  $\alpha > 0$ ). Avec **glmnet()**, nous devons fixer la valeur de  $\alpha$  et la fonction se charge de déterminer la plage de  $\lambda$  à explorer (Remarque: on peut spécifier explicitement la valeur ou la plage de valeurs de  $\lambda$  à essayer si nous le souhaitons).

```
#Régression Elasticnet  
enet3 <- glmnet(XTrain,yTrain,family="binomial",standardize=FALSE,alpha=0.8)  
plot(enet3,xvar="lambda")
```

Nous disposons d'un graphique "Elasticnet path" (Figure 3)...

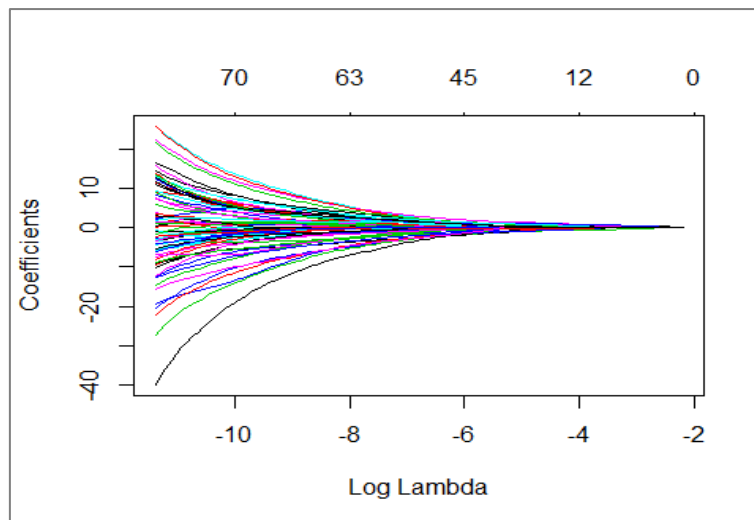


Figure 3 - Elasticnet path

... avec les  $\lambda_i$  suivants :

```
#liste des valeurs de lambda testés
```

```
print(enet3$lambda)
```

```
## [1] 1.122813e-01 1.023065e-01 9.321787e-02 8.493665e-02 7.739111e-02
## [6] 7.051590e-02 6.425146e-02 5.854354e-02 5.334269e-02 4.860387e-02
## [11] 4.428604e-02 4.035179e-02 3.676705e-02 3.350076e-02 3.052465e-02
## [16] 2.781292e-02 2.534210e-02 2.309078e-02 2.103946e-02 1.917037e-02
## [21] 1.746733e-02 1.591558e-02 1.450168e-02 1.321339e-02 1.203955e-02
## [26] 1.096999e-02 9.995446e-03 9.107478e-03 8.298395e-03 7.561189e-03
## [31] 6.889473e-03 6.277432e-03 5.719762e-03 5.211634e-03 4.748647e-03
## [36] 4.326790e-03 3.942410e-03 3.592177e-03 3.273058e-03 2.982289e-03
## [41] 2.717350e-03 2.475948e-03 2.255992e-03 2.055576e-03 1.872964e-03
## [46] 1.706575e-03 1.554968e-03 1.416829e-03 1.290962e-03 1.176276e-03
## [51] 1.071779e-03 9.765651e-04 8.898097e-04 8.107615e-04 7.387357e-04
## [56] 6.731084e-04 6.133113e-04 5.588264e-04 5.091818e-04 4.639475e-04
## [61] 4.227317e-04 3.851774e-04 3.509593e-04 3.197810e-04 2.913726e-04
## [66] 2.654878e-04 2.419026e-04 2.204127e-04 2.008318e-04 1.829905e-04
## [71] 1.667341e-04 1.519219e-04 1.384256e-04 1.261282e-04 1.149233e-04
## [76] 1.047139e-04 9.541138e-05 8.693529e-05 7.921220e-05 7.217521e-05
## [81] 6.576336e-05 5.992113e-05 5.459790e-05 4.974757e-05 4.532813e-05
## [86] 4.130131e-05 3.763221e-05 3.428907e-05 3.124293e-05 2.846739e-05
## [91] 2.593843e-05 2.363413e-05 2.153454e-05 1.962147e-05 1.787835e-05
## [96] 1.629009e-05 1.484292e-05 1.352432e-05 1.232285e-05 1.122813e-05
```

Lorsque nous affichons les  $\lambda_i$  avec le nombre de coefficients non-nuls, nous constatons qu'elasticnet, à la différence de ridge, procède bien à une sélection de variables. Pour la



première valeur de  $\lambda_1 = 0.1228$ , aucune variable n'est sélectionnée (tous les coefficients sont nuls, à l'exclusion de la constante), pour la 2<sup>nde</sup> ( $\lambda_2 = 0.1023$ ), 2 variables sont actives, ..., pour la 15<sup>ème</sup> ( $\lambda_{15} = 0.03052465$ ), il y en a 7, etc.

```
#affichage - nombre de variables sélectionnées vs. Lambda (alpha = 0.8)
```

```
print(cbind(enet3$lambda,enet3$df))
```

```
##           [,1] [,2]
## [1,] 1.122813e-01  0
## [2,] 1.023065e-01  2
## [3,] 9.321787e-02  2
## [4,] 8.493665e-02  2
## [5,] 7.739111e-02  2
## [6,] 7.051590e-02  3
## [7,] 6.425146e-02  4
## [8,] 5.854354e-02  4
## [9,] 5.334269e-02  4
## [10,] 4.860387e-02  4
## [11,] 4.428604e-02  5
## [12,] 4.035179e-02  5
## [13,] 3.676705e-02  5
## [14,] 3.350076e-02  6
## [15,] 3.052465e-02  7
## [16,] 2.781292e-02  11
## [17,] 2.534210e-02  11
## [18,] 2.309078e-02  11
## [19,] 2.103946e-02  11
## [20,] 1.917037e-02  12
## [21,] 1.746733e-02  12
## [22,] 1.591558e-02  13
## [23,] 1.450168e-02  13
## [24,] 1.321339e-02  13
## [25,] 1.203955e-02  14
## [26,] 1.096999e-02  15
## [27,] 9.995446e-03  17
## [28,] 9.107478e-03  17
## [29,] 8.298395e-03  20
## [30,] 7.561189e-03  21
## [31,] 6.889473e-03  22
## [32,] 6.277432e-03  24
## [33,] 5.719762e-03  27
## [34,] 5.211634e-03  33
## [35,] 4.748647e-03  35
## [36,] 4.326790e-03  37
## [37,] 3.942410e-03  40
## [38,] 3.592177e-03  39
## [39,] 3.273058e-03  40
```





```
## [40,] 2.982289e-03 43
## [41,] 2.717350e-03 44
## [42,] 2.475948e-03 45
## [43,] 2.255992e-03 47
## [44,] 2.055576e-03 48
## [45,] 1.872964e-03 49
## [46,] 1.706575e-03 54
## [47,] 1.554968e-03 54
## [48,] 1.416829e-03 55
## [49,] 1.290962e-03 56
## [50,] 1.176276e-03 58
## [51,] 1.071779e-03 58
## [52,] 9.765651e-04 58
## [53,] 8.898097e-04 58
## [54,] 8.107615e-04 57
## [55,] 7.387357e-04 58
## [56,] 6.731084e-04 59
## [57,] 6.133113e-04 58
## [58,] 5.588264e-04 60
## [59,] 5.091818e-04 61
## [60,] 4.639475e-04 61
## [61,] 4.227317e-04 60
## [62,] 3.851774e-04 60
## [63,] 3.509593e-04 60
## [64,] 3.197810e-04 63
## [65,] 2.913726e-04 63
## [66,] 2.654878e-04 63
## [67,] 2.419026e-04 64
## [68,] 2.204127e-04 64
## [69,] 2.008318e-04 66
## [70,] 1.829905e-04 66
## [71,] 1.667341e-04 66
## [72,] 1.519219e-04 66
## [73,] 1.384256e-04 66
## [74,] 1.261282e-04 66
## [75,] 1.149233e-04 66
## [76,] 1.047139e-04 66
## [77,] 9.541138e-05 67
## [78,] 8.693529e-05 67
## [79,] 7.921220e-05 67
## [80,] 7.217521e-05 68
## [81,] 6.576336e-05 68
## [82,] 5.992113e-05 68
## [83,] 5.459790e-05 68
## [84,] 4.974757e-05 70
## [85,] 4.532813e-05 70
## [86,] 4.130131e-05 72
```



```
## [87,] 3.763221e-05 72
## [88,] 3.428907e-05 73
## [89,] 3.124293e-05 73
## [90,] 2.846739e-05 73
## [91,] 2.593843e-05 73
## [92,] 2.363413e-05 73
## [93,] 2.153454e-05 72
## [94,] 1.962147e-05 72
## [95,] 1.787835e-05 73
## [96,] 1.629009e-05 73
## [97,] 1.484292e-05 73
## [98,] 1.352432e-05 73
## [99,] 1.232285e-05 73
## [100,] 1.122813e-05 73
```

Nous pouvons afficher le jeu de coefficients  $\beta^{15}$  correspondant à  $\lambda_{15} = 0.03052465$ .

```
#coefficients pour la 15e valeur de lambda
```

```
print(enet3$beta[,15])
```

```
##          v1          v2          v3          v4          v5
## -0.0814682915 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v6          v7          v8          v9          v10
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v11         v12         v13         v14         v15
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v16         v17         v18         v19         v20
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v21         v22         v23         v24         v25
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v26         v27         v28         v29         v30
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v31         v32         v33         v34         v35
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v36         v37         v38         v39         v40
## 0.0000000000 0.0000000000 0.0000000000 0.6386205987 0.4102639513
##          v41         v42         v43         v44         v45
## 0.0000000000 -0.4871760878 0.0000000000 0.0000000000 0.0000000000
##          v46         v47         v48         v49         v50
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v51         v52         v53         v54         v55
## 0.4125198122 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v56         v57         v58         v59         v60
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##          v61         v62         v63         v64         v65
## 0.0000000000 0.0000000000 0.6904450742 0.0000000000 0.0000000000
##          v66         v67         v68         v69         v70
```



```
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##      v71      v72      v73      v74      v75
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##      v76      v77      v78      v79      v80
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 -0.0001222951
##      v81      v82      v83      v84      v85
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##      v86      v87      v88      v89      v90
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##      v91      v92      v93      v94      v95
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##      v96      v97      v98      v99     v100
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##     v101     v102     v103     v104     v105
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##     v106     v107     v108     v109     v110
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##     v111     v112     v113     v114     v115
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##     v116     v117     v118     v119     v120
## 0.0000000000 0.0000000000 0.0000000000 0.0000000000 0.0000000000
##     v121     v122     v123
## 0.0000000000 0.0000000000 0.0000000000
```

Si l'on s'en tient aux coefficients non-nuls, nous identifions les 7 variables sélectionnées pour ( $\alpha = 0.8$  et  $\lambda = 0.03052465$ ) :

```
#coefficients non-nuls pour la 15e valeur de lambda
print(enet3$beta[abs(enet3$beta[,15])>0,15])
```

```
##      v1      v39      v40      v42      v51
## -0.0814682915 0.6386205987 0.4102639513 -0.4871760878 0.4125198122
##      v63      v80
## 0.6904450742 -0.0001222951
```

**Optimisation de  $\lambda$ .** Ici aussi, nous avons recours à la validation croisée pour détecter la valeur optimale de  $\lambda$  pour  $\alpha = 0.8$ .

```
#validation croisée
cv.enet3 <- cv.glmnet(XTrain,yTrain,family="binomial",type.measure="class",nfolds=
10,alpha=0.8,foldid=cv.ridge2$foldid)
plot(cv.enet3)
```

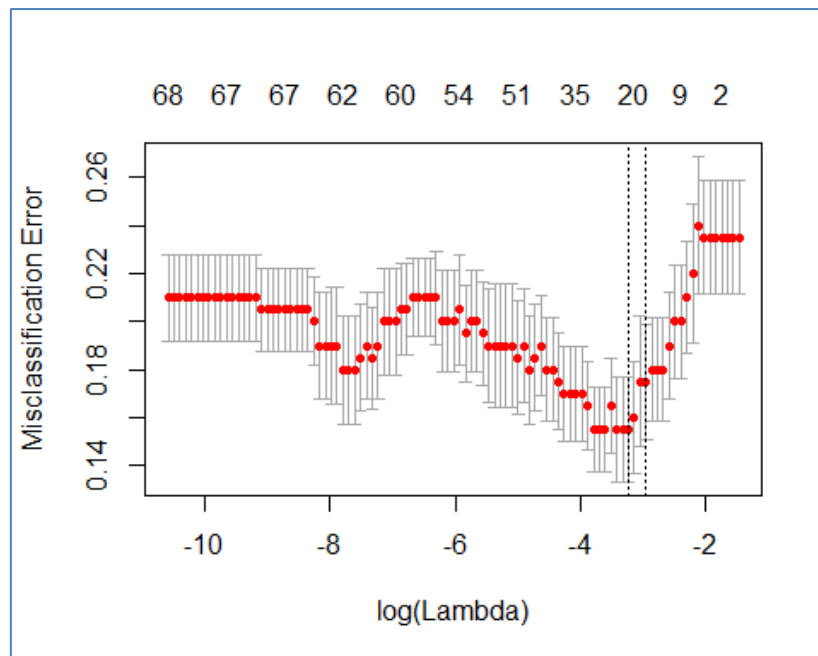


Figure 4 - Elasticnet - Taux d'erreur en validation croisée vs.  $\log(\lambda)$

$\lambda^* = 0.03926356$  minimise l'erreur en validation croisée. Le taux d'erreur du modèle appliqué sur l'échantillon test est de 18.26%.

```
#Lambda min
print(cv.enet3$lambda.min)

## [1] 0.03926356
#prédiction
ye3 <- predict(cv.enet3,XTest,s=c(cv.enet3$lambda.min),type="class")
print(table(ye3))

## ye3
## negative positive
## 26202 4754
#taux d'erreur
print(sum(DTest$classe != ye3)/nrow(DTest))

## [1] 0.1826463
```

**Optimisation conjointe de  $\lambda$  et  $\alpha$ .** Nous avons travaillé à  $\alpha$  fixé dans cette section consacrée à elasticnet ( $\alpha = 0.8$ ). Une extension possible de l'étude serait de tenter d'optimiser conjointement  $\lambda$  et  $\alpha$  de manière à obtenir le taux d'erreur le plus faible en validation croisée. La tâche est simple, il suffit d'englober dans une boucle sur  $\alpha$  le code ci-dessus en veillant à



utiliser des blocs identiques dans la validation croisée pour que les taux d'erreur soient directement comparables. Le résultat s'est pourtant révélé décevant en déploiement sur l'échantillon test, parce qu'à force d'acharnement sur l'échantillon d'apprentissage, même en validation croisée, on finit par faire du surapprentissage. Le modèle « optimal » avec ce processus devient trop spécifique lorsqu'on multiplie les paramètres à manipuler simultanément. Il faudrait introduire une forme de lissage dans l'exploration des solutions pour éviter cet écueil. L'affaire n'est pas triviale.

## 4 Les bibliothèques « tensorflow / keras »

Nous avons présenté ces bibliothèques dans un précédent tutoriel ([Avril 2018](#)). Pour rappel, « Keras » est une surcouche qui permet d'accéder relativement facilement aux fonctionnalités de « tensorflow », particulièrement puissantes, mais particulièrement ésotériques également (pour ne pas dire hermétiques).

### 4.1 Régression logistique sous forme de perceptron simple

La régression logistique proprement dite n'existe pas sous « keras ». Nous passons par l'implémentation d'un perceptron simple avec une fonction de transfert sigmoïde et une fonction de perte correspondant à la log-vraisemblance. En définitive, notre processus d'apprentissage est assimilable à celui de la régression logistique, avec un algorithme d'optimisation spécifique tout simplement.

Nous codons en binaire 0/1 la variable cible, puis nous construisons la structure de réseau.

```
#recoder la cible
y01Train <- ifelse(DTrain$classe=="positive",1,0)

#régression avec descente du gradient, sans coefficient de pénalité
#Bibliothèque Keras
library(keras)

#initialiser la structure
kreg <- keras_model_sequential()
```



```
#ajouter une couche (entrée -> sortie)
kreg %>%
  layer_dense(units=1,input_shape=c(ncol(XTrain)),activation="sigmoid")

#vérification
print(summary(kreg))

## _____
## Layer (type)                Output Shape                Param #
## =====
## dense_1 (Dense)             (None, 1)                   124
## =====
## Total params: 124
## Trainable params: 124
## Non-trainable params: 0
## _____
## NULL
```

Nous avons un perceptron simple, sans couche cachée. Le nombre de coefficients à estimer est 124 c.-à-d.  $p = 123$  explicatives + la constante.

L'instruction **compile()** permet de spécifier la fonction de perte à utiliser (**loss**) et l'algorithme (**optimizer = sgd** : descente de gradient stochastique). Le taux de reconnaissance (**accuracy**) sera utilisé pour suivre l'évolution du processus d'apprentissage.

```
#configuration de L'apprentissage
kreg %>% compile(
  loss = "binary_crossentropy",
  optimizer = "sgd",
  metrics = "accuracy"
)
```

La commande **fit()** lance l'apprentissage proprement dit. Nous indiquons le nombre d'itérations sur la base complète (epochs).

```
#Lancer les calculs
kreg %>% fit(
  x = XTrain,
  y = y01Train,
  epochs = 100
)
```

Deux graphiques de suivi de l'optimisation apparaissent durant l'apprentissage : l'un pour la fonction de coût (**loss**), l'autre pour la mesure de performance (**metrics**) (Figure 5).

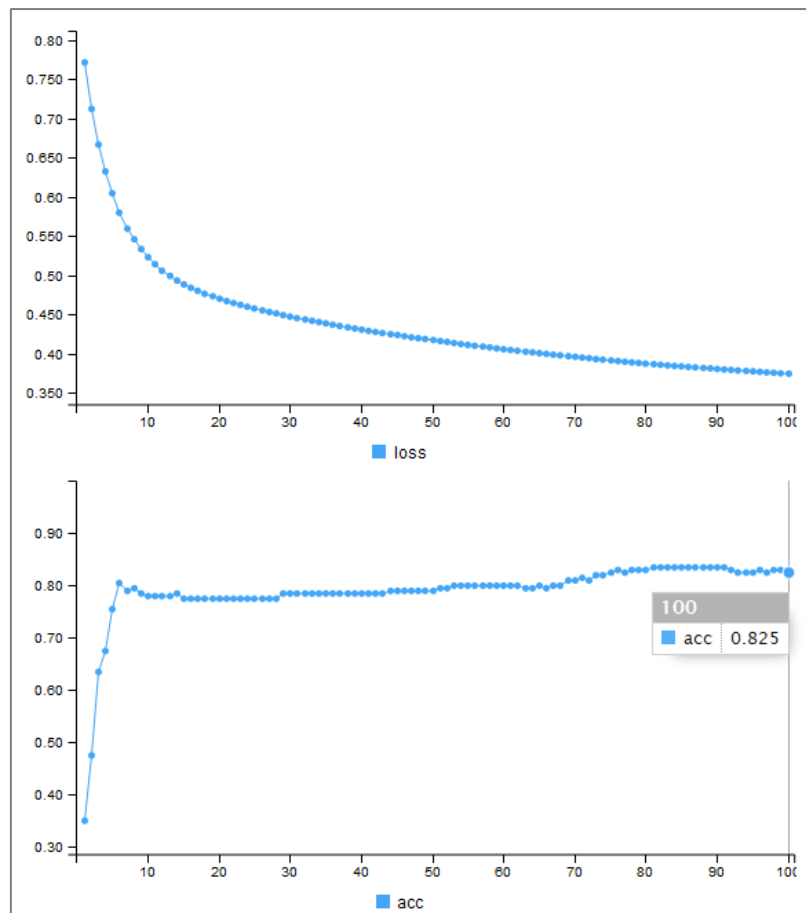


Figure 5 - Suivi du processus d'apprentissage – Package Keras

A l'issue du processus, le taux d'erreur, calculé sur l'échantillon d'apprentissage, est de 17.5% (taux de reconnaissance = 82.5%). Nous verrons ce qu'il en est sur l'échantillon test.

Pour l'heure, inspectons un peu les résultats. Nous récupérons le vecteur des poids synaptiques estimés, qui représentent les coefficients de la régression logistique en fait.

*#récupérer*

```
poids.kreg <- get_weights(kreg)[[1]][,1]
```

Nous calculons le carré de la norme L2 des coefficients. A priori, avec ridge et elasticnet, nous devrions obtenir une valeur plus faible. C'est pour cela qu'on parle de « shrinkage » (rétrécissement) dans la littérature. Notre valeur de référence pour la régression sans



pénalité est  $\|\beta\|_2^2 = 3.192218$  (Remarque: il y a une part d'aléatoire dans les calculs, il se peut que vous obteniez des résultats légèrement différents).

```
#afficher Le carré de Leur norme  
print(sum(poids.kreg^2))
```

```
## [1] 3.192218
```

```
#histogramme des coefficients  
hist(poids.kreg)
```

Une autre manière de considérer les coefficients est d'afficher leur histogramme de fréquence. Nous devrions observer un rétrécissement pour ridge, idem pour elasticnet avec, pour ce dernier, des coefficients nuls.

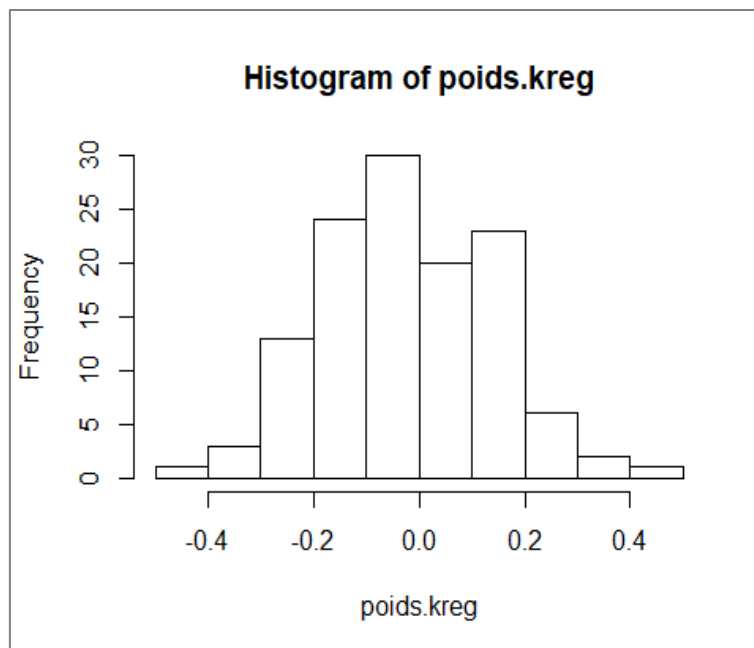


Figure 6 - Histogramme de distribution des coefficients - Régression sans pénalités

La prédiction fournit les scores d'appartenance aux classes. Il s'agit de valeurs réelles comprises entre 0 et 1 puisque nous avons utilisé une fonction de transfert sigmoïde dans le réseau. L'histogramme nous le confirme (Figure 7).

```
#prédiction -> score  
kscore <- kreg %>% predict(XTest)  
hist(kscore)
```



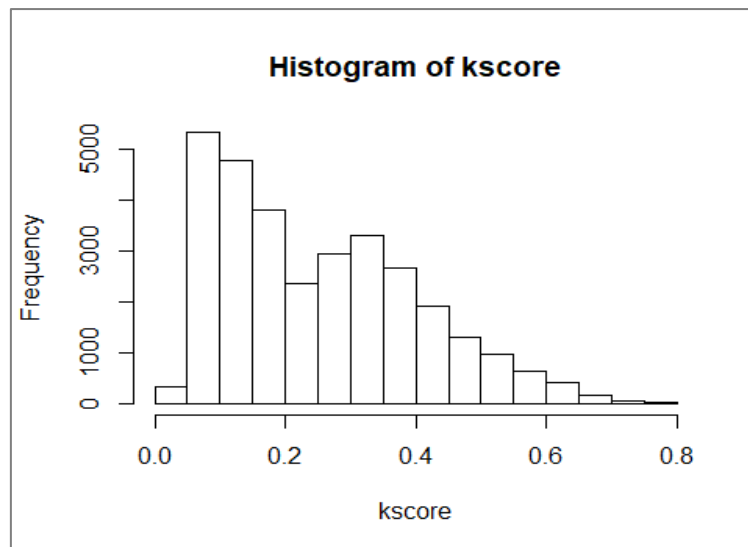


Figure 7 - Histogramme de distribution des scores - Echantillon test - Régression logistique

Il faut comparer le score à la valeur seuil 0.5 pour le convertir en prédiction. Nous constatons qu'il n'y a pas de prédiction systématique d'une des classes...

```
#conversion en classe
ypk <- ifelse(kscore > 0.5, "positive", "negative")
print(table(ypk))

## ypk
## negative positive
## 28711 2245
```

... et le **taux d'erreur est de 19.68%**. Contrairement à **glm()** et **glmnet()**, avec une régression non régularisée, sans précautions particulières, l'outil arrive à engranger de l'information utile pour produire un modèle qui fait mieux que le classifieur par défaut. L'algorithme est solide !

```
#taux d'erreur
print(sum(DTest$classe != ypk)/nrow(DTest))

## [1] 0.1968278
```

## 4.2 Régression Ridge sous « Tensorflow / Keras »

La fonction de coût avec pénalité ressemblerait à ceci sous Keras :



$$J = - \left[ \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} y_i \cdot (\beta_0 + x_i^T \beta) - \log \left( 1 + e^{(\beta_0 + x_i^T \beta)} \right) \right] + \lambda_2 \cdot \|\beta\|_2^2 + \lambda_1 \cdot \|\beta\|_1$$

J'utilise le conditionnel parce que le paramétrage est quand même un peu ambigu sous Keras. On peut définir une fonction de coût global pour l'apprentissage avec la commande **compile()**. En revanche les coefficients de pénalités sont introduites au niveau des couches avec l'instruction **layer\_dense()**. N'ayant qu'une couche reliant la couche à la sortie dans notre réseau, on peut penser que notre équation décrit correctement la grandeur à optimiser. Je suis autrement plus sceptique si nous en avons plusieurs et qu'il nous venait la fantaisie d'introduire des coefficients de pénalité différents d'une couche à l'autre.

Pour l'instant, nous sommes dans un schéma simple, nous spécifions  $\lambda_2 = 0.05$  avec l'option 'kernel\_regularizer'<sup>2</sup> de la commande **layer\_dense()**.

```
#initialiser la structure
kridge <- keras_model_sequential()

#ajouter une couche (entrée -> sortie)
kridge %>%
  layer_dense(units=1,input_shape=c(ncol(XTrain)),activation="sigmoid",
              kernel_regularizer = regularizer_l2(0.05))

#configuration de l'apprentissage
kridge %>% compile(
  loss = "binary_crossentropy",
```

---

<sup>2</sup> La documentation de Keras n'est pas très claire non plus à ce sujet. Il y a deux types de régularisations possibles sur un `layer_dense()` : "kernel", où l'on semble chercher à harmoniser les poids synaptiques, cela correspond à ce que nous souhaitons faire ; "activation", où il s'agirait plutôt d'harmoniser les sorties des neurones de la même couche (s'il y en a plusieurs). Il n'est nullement fait mention de la régression ridge ou lasso dans la documentation... actuelle, pas l'ancienne. En cherchant attentivement sur le web, j'ai eu accès à la documentation de Keras 1.2.2 (<https://faroit.github.io/keras-docs/1.2.2/regularizers/>), où une autre option était proposée, 'weight regularizer'. Elle semble obsolète aujourd'hui. On ne sait pas très bien s'il y a une correspondance, et jusqu'à quel point, entre cette dernière et 'kernel regularizer' que nous utilisons (Version 2.1.5, Mai 2018).



```
optimizer = "sgd",
metrics = "accuracy"
)

#lancer les calculs
kridge %>% fit(
  x = XTrain,
  y = y01Train,
  epochs = 100
)

#récupérer les poids
poids.kridge <- get_weights(kridge)[[1]][,1]

#afficher le carré de leur norme
print(sum(poids.kridge^2))

## [1] 1.213311
```

Le « shrinkage » (rétrécissement) porte bien son nom, le carré de la norme a été réduit à  $\|\beta_{\lambda_2=0.05}\|_2^2 = 1.213311$  contre  $\|\beta\|_2^2 = 3.192218$ . L'étendue des coefficients est réduite comme nous le constatons avec l'histogramme de distribution des coefficients (Figure 8).

```
#histogramme des coefficients
hist(poids.kridge)
```

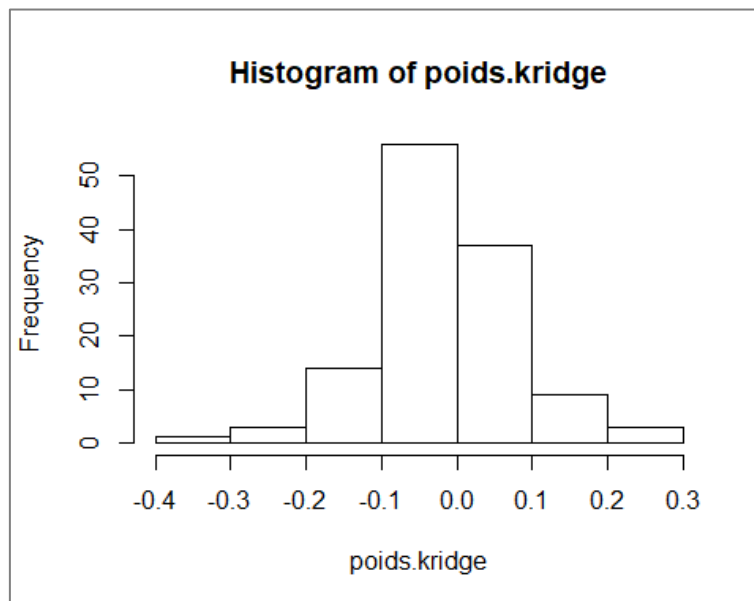


Figure 8 - Histogramme de distribution des coefficients - Régression ridge

Mais est-ce à juste titre ? Voyons si les prédictions sont meilleures en déploiement.



```
#prédiction -> score
ridge_score <- kridge %>% predict(XTest)

#conversion en classe
ypkridge <- ifelse(ridge_score > 0.5,"positive","negative")
print(table(ypkridge))

## ypkridge
## negative positive
##      30012      944
#taux d'erreur
print(sum(DTest$classe != ypkridge)/nrow(DTest))

## [1] 0.2168239
```

Non finalement. Le modèle n'est pas meilleur.

**Remarque:** Bien sûr, à l'instar du travail effectué à l'aide de « glmnet », nous pouvons essayer d'optimiser  $\lambda_2$  en validation croisée. Je n'ai pas trouvé d'outil dédié sous Keras (on n'en parle pas dans la [documentation](#)), mais on peut y palier sans difficulté avec un peu de programmation.

### 4.3 Régression elasticnet sous « tensorflow / keras »

Nous spécifions  $\lambda_1 = 0.01$  et  $\lambda_2 = 0.005$  pour la régression elasticnet. Nous donnons plus d'importance à  $\lambda_1$  pour avoir un comportement proche de Lasso.

```
#initialiser la structure
kenet <- keras_model_sequential()

#ajouter une couche (entrée -> sortie) -- elasticnet
kenet %>%
  layer_dense(units=1,input_shape=c(ncol(XTrain)),activation="sigmoid",
              kernel_regularizer = regularizer_l1_l2(l1=0.01,l2=0.005))

#configuration de l'apprentissage
kenet %>% compile(
  loss = "binary_crossentropy",
  optimizer = "sgd",
  metrics = "accuracy"
)
```



```
#Lancer les calculs
kenet %>% fit(
  x = XTrain,
  y = y01Train,
  epochs = 100
)

#récupérer Les poids
poids.kenet <- get_weights(kenet)[[1]][,1]

#afficher Le carré de leur norme
print(sum(poids.kenet^2))

## [1] 1.562286

#histogramme des coefficients
hist(poids.kenet)
```

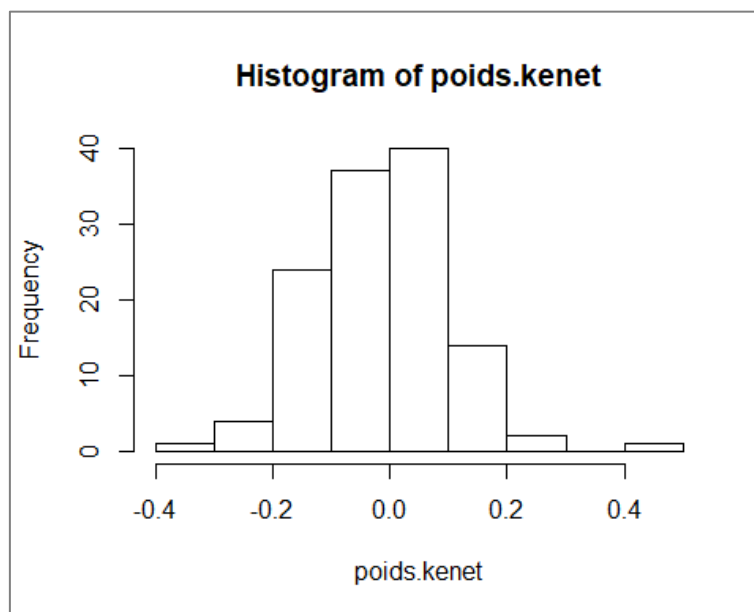


Figure 9 - Distribution des coefficients - Régression elasticnet

```
#prédiction -> score
enet_score <- kenet %>% predict(XTest)

#conversion en classe
ypkenet <- ifelse(enet_score > 0.5, "positive", "negative")
print(table(ypkenet))

## ypkenet
## negative positive
## 29172 1784
```



```
#taux d'erreur
print(sum(DTest$classe != ypkenet)/nrow(DTest))
## [1] 0.2038377
```

Avec le paramétrage spécifié ( $\lambda_1 = 0.01$  et  $\lambda_2 = 0.005$ ), le taux d'erreur est intermédiaire entre l'approche ridge ( $\lambda_2 = 0.05$ ) et le modèle sans pénalité ( $\lambda_1 = 0$  et  $\lambda_2 = 0$ ).

## 5 Récapitulatif des résultats

Nous avons beaucoup tenté avec de multiples outils dans ce tutoriel. Voici un tableau récapitulatif des performances en test selon les packages et paramétrages utilisés, sachant que le taux d'erreur du classifieur par défaut est 24%.

Outil	Sans régularisation	Ridge	Elasticnet
glm (« stats »)	Echec	-	-
« glmnet »	31.5%	17.7% ( $\lambda = 0.29$ )	18.3% ( $\lambda = 0.04, \alpha = 0.8$ )
Tensorflow / keras	19.6%	21.7% ( $\lambda_2 = 0.05$ )	20.4% ( $\lambda_1 = 0.01, \lambda_2 = 0.005$ )

Les écarts ne sont absolument pas négligeables sur un échantillon test de 30956 observations. Manifestement, les paramètres pèsent fortement sur les résultats. Les outils tels que « glmnet » qui proposent des solutions clés en main pour déterminer semi-automatiquement les bonnes plages de valeurs des paramètres de régularisation se révèlent décisifs dans ce contexte.

## 6 Conclusion

L'objectif de ce tutoriel était de montrer la mise en œuvre des régressions ridge et elasticnet sous R, un précédent document étant consacré à la [régression lasso sous Python](#). Les outils étudiés nous ont permis de mener à bien les tâches que nous nous sommes assignées. Nous nous sommes focalisés sur la manipulation des coefficients de pénalités qui pèsent directement sur les propriétés de régularisation des approches.



Mais l'étude de la documentation montre que les fonctions sont en réalité bardées de tous un tas de paramètres censés peser sur la qualité et vitesse de convergence. Malgré une lecture assidue, on a du mal à cerner concrètement leur influence sur la qualité prédictive des modèles. Et je n'ai pas vu de tutoriels, même en anglais, qui en fassent le tour et explicitent leurs rôles sur des exemples concrets (un peu quand même dans les [annexes](#) du site de « glmnet »). Il reste du travail encore de ce côté-là...

## 7 Références

- (HASTIE & QIAN, 2016) Hastie T., Qian J., « Glmnet Vignette », Septembre 2016 ; [https://web.stanford.edu/~hastie/glmnet/glmnet\\_beta.html](https://web.stanford.edu/~hastie/glmnet/glmnet_beta.html)
- Friedman J., Hastie T., Tibshirani R., Simon N., Narasimhan B., Qian J., « glmnet: Lasso and Elastic-Net Regularized Generalized Linear Models », version 2.0-16, Avril 2018 ; <https://cran.r-project.org/package=glmnet>
- (FRIEDMAN et al., 2010) Friedman J., Hastie T., Tibshirani R., « [Regularization Paths for Generalized Linear Models via Coordinate Descent](#) », in Journal of Statistical Software, Volume 33, Issue 1, January 2010.
- (RAK, 2018) « Ridge, Lasso, Elasticnet - Diapos », Mai 2018 ; <http://tutoriels-data-mining.blogspot.fr/2018/05/ridge-lasso-elasticnet.html>
- Keras Documentation. « Keras : The Python Deep Learning Library » ; <https://keras.io/>
- J.J. Allaire, F. Chollet, RStudio, Google, Y. Tang, D. Falbel, W. Van Der Bijl, M. Studer, « keras: R Interface to 'Keras' », version 2.1.6, Avril 2018 ; <https://cran.r-project.org/package=keras>