Outils d'optimisation sous R

Tutoriel Tanagra

1 Introduction

J'utilise quasiment toujours le tableur Excel pour disséquer les algorithmes de machine learning. Il n'y a rien de mieux je trouve pour décortiquer les formules. On ne peut pas rentrer des commandes au petit bonheur la chance, nous sommes obligés de tout comprendre pour pouvoir tout décomposer. Comme une grande partie des méthodes revient à optimiser une fonction de perte (ou de gain), je m'appuie alors sur le solveur. J'obtiens souvent des résultats satisfaisants, comme par exemple dans mon ouvrage – qui servira de référence – consacré à la "Pratique de la régression logistique" où l'on maximise la log-vraisemblance (LIVRE, section 1.4, pages 15 à 19).

Je me suis demandé s'il existait un équivalent du solveur sous R. En cherchant un peu, je me suis rendu compte que oui, il s'agit de la fonction `optim()` du package "stats", installé et chargé par défaut sous R. Tout comme son homologue sous Excel, il peut fonctionner avec seulement une fonction objectif et un vecteur de paramètres. Mais il peut aller plus loin, nous pouvons lui fournir d'autres informations pour qu'il soit plus efficace. Il sait produire également des résultats additionnels nécessaires à l'inférence statistique lorsque nous travaillons sur les algorithmes de régression par exemple.

Dans ce tutoriel, nous montrons l'utilisation des fonctions `optim()` et `optimHess()` pour la programmation de la régression logistique. Nous comparerons les résultats d'une part avec les sorties de la fonction `glm()` de R, d'autre part avec les fruits d'une petite implémentation maison de l'algorithme de Newton-Raphson.

2 Données et résultats de la glm() de R

Nous reprenons les données de notre ouvrage de référence (LIVRE, figure 0.1, page 2). Nous cherchons à expliquer l'occurrence d'une maladie cardiaque à partir des caractéristiques

des patients : l'âge, le taux max et l'angine (de poitrine). Pour faciliter les manipulations, la cible COEUR a été codée en (0/1, absence/présence), et nous avons adjoint la colonne de constante 1 dans le bloc des explicatives.

coeur	const	age	taux_max	angine
1	1	50	126	1
1	1	49	126	0
1	1	46	144	0
1	1	49	139	0
1	1	62	154	1
1	1	35	156	1
0	1	67	160	0
0	1	65	140	0
0	1	47	143	0
0	1	58	165	0
0	1	57	115	1
0	1	59	145	0
0	1	44	175	0
0	1	41	153	0
0	1	54	152	0
0	1	52	169	0
0	1	57	168	1
0	1	50	158	0
0	1	44	170	0
0	1	49	171	0

Nous importons le fichier "**calcul_avec_optim_R.xlsx**" sous R.

```
#changer de répertoire
setwd("... votre dossier ...")

#charger les données
library(xlsx)
D <- read.xlsx("calcul_avec_optim_R.xlsx",sheetIndex=1)
str(D)

## 'data.frame':    20 obs. of  5 variables:
##  $ coeur   : num  1 1 1 1 1 1 0 0 0 0 ...
##  $ const   : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ age     : num  50 49 46 49 62 35 67 65 47 58 ...
##  $ taux_max: num  126 126 144 139 154 156 160 140 143 165 ...
##  $ angine  : num  1 0 0 0 1 1 0 0 0 0 ...
```

Nous disposons de 20 observations et 5 variables.

Nous lançons la régression logistique (family = binomial) avec la commande `glm()` du package "stats". Nous affichons les coefficients estimés.

```r
#régression logistique
lr <- glm(coeur ~ age+taux_max+angine,data=D,family=binomial)

#coefficients
print(lr$coefficients)
```

```
## (Intercept)         age    taux_max      angine
##  14.4937905  -0.1256341  -0.0635603   1.7790129
```

La valeur de la fonction objectif (log-vraisemblance) peut être dérivée de la déviance (et inversement, LIVRE, page 17).

```r
#log-vraisemblance
print(-lr$deviance/2.0)
```

```
## [1] -8.308844
```

`glm()` fait référence auprès des statisticiens. Notre propos dans ce tutoriel est d'utiliser les outils d'optimisation de R pour essayer de retrouver ces résultats, en leur fournissant des informations de plus en plus riches en entrée.

3 Optimisation de la log-vraisemblance

3.1 Etape préparatoire

Nous isolons l'endogène (variable cible, y) et les exogènes (variables explicatives, X_j) dans des structures distinctes.

```r
#structures intermédiaires
endog <- D[,1]
exog <- as.matrix(D[,2:5])
```

Puis nous créons une fonction d'initialisation du vecteur des coefficients à estimer. Elle est paramétrable afin que l'expérimentation soit reproductible.

```r
#fonction pour initialisation du vecteur de paramètres
initialiser <- function(seed=1,ncol=4){
  #initialisation du générateur
  set.seed(seed)
  #valeurs de départ des coefficients
  coef <- runif(ncol)-0.5
  return(coef)
}
```

3.2 Optimisation avec la fonction de log-vraisemblance

La log-vraisemblance à maximiser s'écrit comme suit (LIVRE, section 1.4, les notations sont un peu différentes dans ce tutoriel pour être en accord avec le programme R) :

$$LL = \sum_{i=1}^{n} y_i \ln \pi_i + (1 - y_i) \ln \pi_i$$

Où (n = 20) est la taille de l'échantillon, π_i ($proba_i$) est la probabilité conditionnelle d'appartenir à (COEUR = 1), avec

$$\pi_i = \frac{1}{1 + e^{-v_i}}$$

v_i est le LOGIT calculé à partir des coefficients estimés.

$$v_i = \hat{a}_0 + \hat{a}_1 x_{i,1} + \cdots + \hat{a}_p x_{i,p}$$

Sous R, la fonction de log-vraisemblance s'écrit comme suit :

```
#fonction pour le calcul de la log-vraisemblance
myLL <- function(param,Y,X){
  #calcul de la combinaison linéaire, puis proba (pi)
  #pour chaque individu de la matrice X
  proba <- apply(X,1,function(x){v <- sum(param*x);return(1.0/(1.0+exp(-v)))})
  #log-vraisemblance
  LL <- sum((Y*log(proba)+(1.0-Y)*log(1.0-proba)))
  return(LL)
}
```

Avec les coefficients de départ fournis par la fonction `initialiser()`, nous aurions par exemple LL = -62.30662.

```
#initialisation
coef_depart <- initialiser()

#calcul de la vraisemblance avec ces coeffs
print(myLL(coef_depart,endog,exog))

## [1] -62.30662
```

Nous pouvons maintenant faire appel à la fonction d'optimisation `optim()`, nous lui passons comme paramètres :

- (par = coef_depart) représente le vecteur des coefficients de départ.

- (fn = myLL) représente la fonction *callback* correspondant au critère à optimiser.
- (method = "BFGS") est l'algorithme d'optimisation de Broyden–Fletcher–Goldfarb–Shanno. Dans cette expérimentation, R se charge lui-même d'approximer numériquement le vecteur gradient (dérivée partielle première) et la matrice hessienne (dérivée partielle seconde).
- Y et X correspondent à l'endogène et aux exogènes, comprenant la constante.
- (control = settings) nous sert à préciser un processus de maximisation (fnscale=-1).

```
#paramètres pour l'optimisation
#fnscale = -1 ==> maximisation
settings <- list(fnscale=-1)

#optimisation
res1 <-
stats::optim(par=coef_depart,fn=myLL,method="BFGS",Y=endog,X=exog,control=settings)
print(res1)

## $par
## [1] 14.66514615 -0.12625831 -0.06449995  1.77773116
##
## $value
## [1] -8.309124
##
## $counts
## function gradient
##      106       29
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Nous notons que la fonction objectif a été appelée 106 fois, que le vecteur gradient a été estimé 29 fois via une approximation par différence finie.

Au final, l'optimum obtenu est en deçà de celui de glm() (-8.309124 vs. -8.308844), les coefficients sont sensiblement différents à partir de la 3ème décimale. Ne fournir à `optim()` que la fonction objectif et le laisser estimer le gradient ne lui permet pas d'être efficace dans notre contexte.

4 Optimisation avec la formule du gradient

4.1 Fonction pour le calcul du vecteur gradient

Dans cette section, nous essayons de tirer profit du paramètre "gr" en lui passant une fonction calculant le gradient (LIVRE, section 1.5.2, page 21).

Son expression est la suivante :

$$\nabla a_j = \sum_{i=1}^{n} (y_i - \pi_i) x_{i,j}$$

Que nous traduisons sous forme de fonction comme suit :

```
#fonction pour calcul du gradient
myGradient <- function(param,Y,X){
  #calcul de la combinaison linéaire, puis pi (proba)
  #pour chaque individu de la matrice X
  proba <- apply(X,1,function(ligne){v<-sum(param*ligne);return(1.0/(1.0+exp(-v)))})
  #calcul du vecteur gradient
  gradient <- apply(X,2,function(colonne){sum((Y-proba)*colonne)})
  return(gradient)
}
```

Elle prend en entrée trois paramètres : le vecteur des coefficients de la régression logistique (param), le vecteur de l'endogène (Y) et la matrice des exogènes (X). Elle renvoie le vecteur des dérivées partielles première dans le voisinage concerné.

4.2 Vecteur gradient à l'optimum

Le gradient correspond au vecteur nul (tous ses éléments sont nuls) lorsque nous sommes à l'optimum. Voyons ce qu'il en est lorsque nous passons en paramètres les coefficients fournis par `glm()`. Cette configuration peut constituer une règle d'arrêt de l'algorithme d'optimisation d'ailleurs.

```
#valeur du gradient à l'optimum GLM
gradGlm <- myGradient(lr$coefficients,endog,exog)
print(gradGlm)
```

```
##         const           age      taux_max        angine
## -7.777910e-13 -1.194831e-10 -1.109712e-10  2.971012e-12
```

Manifestement, nous ne sommes pas loin de la vérité. Nous calculons la norme du vecteur par acquit de conscience.

```
#norme du vecteur gradient
print(sqrt(sum(gradGlm^2)))

## [1] 1.630958e-10
```

Voyons ce qu'il en est maintenant pour la solution produite par le premier appel d'`optim()`.

```
#valeur du gradient à l'optimum res1
gradRes1 <- myGradient(res1$par,endog,exog)
print(gradRes1)

##         const           age      taux_max        angine
## -2.863177e-03 -9.437118e-02  1.345605e-01  2.056734e-05

#sa norme
print(sqrt(sum(gradRes1^2)))

## [1] 0.1643796
```

La solution est de mauvaise qualité, il n'y a plus de doute là-dessus.

4.3 Optimisation incluant la fonction gradient

Nous lançons de nouveau l'optimisation exactement dans les mêmes conditions mais avec le paramètre supplémentaire (gr = myGradient). En prodiguant la vraie valeur du gradient à chaque étape, nous espérons obtenir un résultat de meilleure qualité.

```
#départ vecteur de paramètres
coef_depart <- initialiser()

#optimisation
res2 <-
stats::optim(par=coef_depart,fn=myLL,gr=myGradient,method="BFGS",Y=endog,X=exog,contr
ol=settings)
print(res2)

## $par
## [1] 14.51192271 -0.12563794 -0.06367956  1.77834580
##
## $value
## [1] -8.308849
##
## $counts
## function gradient
##       67       28
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Le nombre d'appels du gradient, notre fonction cette fois-ci, est peu modifié. En revanche, la fonction objectif est moins sollicitée (67 vs. 160 précédemment).

En tous les cas, nous nous rapprochons de la vérité (-8.308844 pour `glm`). En calculant le gradient dans ce nouveau voisinage…

```
#vérification du vecteur gradient de la solution
gradRes2 <- myGradient(res2$par,endog,exog)
print(gradRes2)

##         const           age      taux_max        angine
## -3.976598e-04 -2.202059e-02  1.434979e-02  5.946149e-05

#et de sa norme
print(sqrt(sum(gradRes2^2)))

## [1] 0.02628658
```

… nous constatons malgré tout que nous ne sommes pas vraiment sur la crête de l'optimum.

5 Optimisation exploitant la matrice hessienne

5.1 Approximation de la matrice hessienne

La matrice hessienne H (matrice des dérivées partielles secondes, LIVRE, section 1.5.3) est l'information supplémentaire qui devrait nous permettre de cheminer plus efficacement vers l'optimum, via l'algorithme de Newton-Raphson (LIVRE, section 1.5, page 20 et suivantes).

R, à travers la fonction `optimHess()` peut nous en fournir une approximation numérique, qu'elle utilise durant l'optimisation avec BFGS d'ailleurs : *"L'idée principale de cette méthode (BFGS) est d'éviter de construire explicitement la matrice hessienne et de construire à la place une approximation de l'inverse de la dérivée seconde de la fonction à minimiser, en analysant les différents gradients successifs."*(Wikipedia)

Dans le cadre de la régression logistique, H revêt une importance particulière puisque son inverse correspond à la matrice de variance covariance des coefficients estimés.

Voyons ce que donne `optimHess()` dans le voisinage de la solution produite par `glm()`.

```
#approximation de la matrice hessienne
Happrox <- optimHess(par=lr$coefficients,fn=myLL,Y=endog,X=exog)
print(Happrox)

##              (Intercept)          age    taux_max      angine
## (Intercept)   -2.6139657   -130.24123   -386.3509  -0.6468653
## age         -130.2412348  -6615.84442 -19215.7060 -34.5878407
## taux_max    -386.3509452 -19215.70599 -57736.0595 -94.1752980
## angine        -0.6468653    -34.58784    -94.1753  -0.6468653
```

5.2 Calcul de la matrice hessienne

Puisque nous disposons de la formule de la "vraie" matrice (LIVRE, section 1.5.3). Nous programmons une fonction dédiée…

```
#calcul de la matrice Hessienne
myHessian <- function(param,Y,X){
  #calcul de la combinaison linéaire, puis pi (proba)
  #pour chaque individu de la matrice X
  proba <- apply(X,1,function(x){v <- sum(param*x);return(1.0/(1.0+exp(-v)))})
  #matrice W -- diagonale avec proba*(1-proba)
  W <- matrix(0,nrow=nrow(X),ncol=nrow(X))
  diag(W) <- proba*(1.0-proba)
  #matrice Hessienne
  H <- t(X)%*%W%*%X
  return(-1.0*H)
}
```

… et nous calculons la matrice hessienne "exacte" dans le voisinage de la solution de `glm()`.

```
#matrice hessienne avec la bonne formule
Hright <- myHessian(lr$coefficients,Y=endog,X=exog)
print(Hright)

##                 const          age     taux_max      angine
## const      -2.6139657   -130.23795   -386.29150  -0.6468653
## age      -130.2379478  -6615.33264 -19210.81492 -34.5851709
## taux_max -386.2915033 -19210.81492 -57709.00513 -94.1150404
## angine     -0.6468653    -34.58517    -94.11504  -0.6468653
```

Pour évaluer la qualité de l'approximation, nous opposons ces deux matrices (approchée et exacte) :

```
#écart
ecartH <- sum((Happrox-Hright)^2)
print(ecartH)

## [1] 780.0617
```

Manifestement, l'approximation n'est pas de très bonne qualité, expliquant en partie la qualité relative de la fonction `optim()` dans la maximisation de la log-vraisemblance.

5.3 Algorithme de Newton-Raphson

Puisque nous disposons des fonctions pour calculer l'objectif, le vecteur gradient et la matrice hessienne, nous sommes en mesure de programmer l'algorithme de Newton-Raphson (LIVRE, section 1.5, page 20). La petite fonction ci-dessous est assez fruste (le calcul de π_i est répété plusieurs fois par exemple) mais fait le job. J'émettrai plusieurs remarques :

- Il est fortement conseillé de centrer et surtout réduire les variables. C'est ce que je fais dans la première partie de la procédure. La constante n'est pas incluse dans ce processus bien sûr.
- Après coup, il faut dé-standardiser les coefficients obtenus en utilisant les moyennes et écarts-types calculés précédemment. La formule est très simple pour les coefficients des variables, elle est plus complexe pour la constante.
- La convergence est établie lorsque l'écart entre deux valeurs successives de la log-vraisemblance est inférieur à 1e-8.
- Quoiqu'il en soit, si le nombre d'itérations est supérieur à 100 (paramétrable), le processus est stoppé.
- La fonction renvoie en sortie la log-vraisemblance, le vecteur des coefficients, et le nombre d'itérations.

```
#algorithme de newton-raphson
newton_raphson <- function(par,Y,X,maxiter=100){
  #moyennes des variables
  moy <- colMeans(X[,2:ncol(X)])
  #écarts-type
  et <- apply(X[,2:ncol(X)],2,sd)
  #centrer et réduire, pas la constante bien sûr
  Z <- cbind(X[,1],scale(X[,2:ncol(X)],center=moy,scale=et))
  #coefficient
  coef <- par
  #Log-vraisemblance
  LL <- myLL(coef,Y,Z)
  #compteur d'itérations
  iter <- 0
  #boucler jusqu'à convergence
  repeat{
    #compteur d'itération
    iter <- iter + 1
    #calcul du gradient
    grad <- myGradient(coef,Y,Z)
```

```
    #calcul de la matrice Hessienne
    hess <- myHessian(coef,Y,Z)
    #inversion de la matrice Hessienne
    invHess <- solve(hess)
    #màj des coefs
    newCoef <- coef - invHess%*%grad
    #recalcul de la LL
    newLL <- myLL(newCoef,Y,Z)
    #test de continuation
    #écart faible de l'évolution de la LL?
    diff <- (newLL - LL)
    if (diff < 1e-8 || iter > maxiter){
      #convergence - on sort de la boucle
      break
    } else
    {
      #màj. coef
      coef <- newCoef
      #maj. LL
      LL <- newLL
    }
  }
  #recalculer les coefficients - dénormalisation
  #la constante
  coef[1] <- newCoef[1]+sum(sapply(1:3,function(j){-newCoef[j+1]*moy[j]/et[j]}))
  #les autres coefs
  for (j in 2:ncol(X)){coef[j] <- newCoef[j]/et[j-1]}
  #renvoyer les résultats
  return(list(loglik=newLL,coefficients=coef,iterations=iter))
}
```

Nous pouvons maintenant lancer notre fonction en lui passant le même vecteur de coefficient de départ que lors des appels d'`optim()`.

```
#valeur de départ des coefficients
coef_depart <- initialiser()

#lancer les calculs
resNR <- newton_raphson(coef_depart,endog,exog)
print(resNR)

## $loglik
## [1] -8.308844
##
## $coefficients
##                [,1]
##          14.4937905
## age      -0.1256341
## taux_max -0.0635603
## angine    1.7790129
##
## $iterations
## [1] 5
```

5 itérations (seulement) suffisent pour aboutir à la convergence. Les résultats semblent être du même ordre que ceux de `glm()` de R. Voyons cela en calculant le gradient.

```
#valeur du gradient
gradNR <- myGradient(resNR$coefficients,endog,exog)
print(gradNR)

##         const           age      taux_max        angine
## -1.815512e-10 -1.814390e-08 -2.805316e-08  3.098387e-10

#norme du gradient
print(sqrt(sum(gradNR^2)))

## [1] 3.341123e-08
```

Oui, les performances sont comparables.

6 Conclusion

L'objectif de ce tutoriel était d'étudier le mode opératoire et le comportement de la fonction d'optimisation `optim()` de R, qui s'apparente à bien des égards au solveur du tableur Excel. Pour illustrer mon propos, je me suis appuyé sur l'optimisation de la log-vraisemblance pour la régression logistique. L'outil fonctionne, pas de doute là-dessus, mais les performances ne paraissent pas très bonnes, en partie je pense parce que je n'ai pas cherché à explorer les très nombreuses options qu'il propose. Nous pouvons notamment choisir l'algorithme d'optimisation, être plus exigeant sur les conditions de convergence, affiner les approximations du gradient, etc. (Voir la documentation)

Enfin, à titre de curiosité, j'ai mis dans la seconde feuille du classeur "**calcul_avec_optim_R.xlsx**" qui accompagne ce document le paramétrage et les résultats du Solveur d'Excel sur ce même problème. Rappelons qu'il ne prend en compte que la fonction objectif, il ne dispose ni du gradient, encore moins de la matrice hessienne c.-à-d. nous sommes dans une situation analogue au premier scénario d'optimisation (section 3.2). On se rend compte qu'il se comporte plutôt bien.

7 Références

[LIVRE] R. Rakotomalala, "Pratique de la régression logistique", version 2.0, mai 2017.