Université Lumière Lyon 2

Maïté Garcia - Pierre Lejeail - Jean-Luc Yongwe

Master 2 SISE - Semestre 1 - Année 2016-2017

Package Wrap :

Sélection de variables par méthodes wrapper

Cours :
Programmation R

Enseignant :
Ricco Rakotomalala

Rendu :
19 décembre 2016

Table des matières

1 Introduction

Dans le cadre de l'enseignement de programmation R dispensé par Monsieur Ricco Rakotomalala, nous avons créé un package intégrant la méthode de sélection de variables wrapper applicable sur une série de méthodes d'apprentissage supervisé (régression logistique, analyse discriminante linéaire et quadratique, régression linéaire, Support Vector Machine (SVM), classification naïve bayésienne et random forest).

Nous avons implémenté les approches de sélections ascendantes (sélection forward), descendantes (sélection backward) ainsi que l'algorithme génétique.

Notre package porte le nom de "`wrap`". Il peut être diffusé et installé de façon autonome. Il inclut une documentation en anglais. Nous avons utilisé le package microbenchmark pour optimiser le code.

Dans ce rapport nous allons aborder l'état de l'art dans la littérature sur la méthode wrapper. Puis, nous expliciterons l'architecture de notre programme en mettant en évidence les fonctions principales. Nous décrirons ensuite à travers un tutoriel comment intégrer un wrapper dans une analyse avec notre package. Nous terminerons par des propositions d'améliorations et une conclusion sur notre travail.

2 État de l'art

La sélection de features est un élément de plus en plus important dans le domaine du data mining avec la popularité du Big Data (Gowda et al., 2010). D'une part, ces techniques simplifient les jeux de données, facilitant l'estimation des paramètres du modèle, et d'autre part, permettent de construire le meilleur modèle (Ron and George, 1997), selon les critères de sélection choisis.

Cette sélection de variable peut être réalisée en appliquant un **filtre** qui ne conserve que les variables satisfaisant une certaine condition (e.g. corrélation supérieure à 0,5). Cette technique, bien qu'arbitraire et peu précise, reste très populaire, car elle est très rapide et permet donc d'éliminer facilement de nombreuses variables dans les grands jeux de données (Hall and Holmes, 2003). En effet, il n'est pas nécessaire d'estimer le modèle pour pouvoir filtrer les variables, ce qui fait de cette technique un candidat de choix pour élaguer une première fois les données.

Au contraire des filtres, les **wrappers** répètent l'apprentissage du modèle en modifiant les paramètres à chaque itération jusqu'à converger vers un optimum (Ron and George, 1997). Parmi les algorithmes balayant les différents sous-ensembles de paramètres à tester, on peut citer deux grandes catégories de wrapper :

— les algorithmes greedy optimisant pas à pas un indicateur

— les algorithmes stochastiques cherchant aléatoirement la meilleure solution

Parmi les algorithmes greedy, on trouve notamment backward, forward et HillClimbing (combinant le wrapper avec un filtre) (Jonathan et al., 2003). La méthode **backward** est un algorithme qui, partant de l'ensemble des variables explicatives, retire la variable la plus mauvaise à chaque itération selon un critère d'évaluation du modèle (e.g. taux d'erreur). D'une façon similaire, la méthode **forward** part de l'ensemble vide et ajoute la variable explicative la meilleure selon un critère d'évaluation du modèle. Les algorithmes backward et forward sont "greedy" puisqu'ils cherchent à trouver la meilleure solution parmi toutes, et sont donc très sensibles à un nombre de features élevé. Ils sont également sensibles aux optima locaux qui peuvent provoquer un arrêt prématuré avant de trouver l'optimum global.

Les méthodes stochastiques, plus complexe que les précédentes, semblent plus efficaces d'après Grzegorz (2010). En effet, toutes les méthodes stochastiques, à l'exception de la recherche aléatoire, proposent de meilleurs résultats en moyenne que les approches déterministes forward et backward.

L'**algorithme génétique** est une méthode stochastique dite "évolutionnaire" car elle utilise le concept de la sélection naturelle provenant de la théorie de l'évolution. Il est appliqué à une population de solutions potentielles au problème donné (sous-ensemble de features). La solution est approchée par "bonds" successifs. Autrement dit on réduit l'espace de recherche à chaque itération en lui attribuant des bornes supérieures et inférieures de plus en plus restreintes (Rebaïne, 2005).

À partir de toutes les ressources bibliographiques qui nous ont permis de nous approprier les approches wrapper, nous avons réalisé un package `R` impliquant différentes stratégies d'applications des méthodes wrappers et méthodes statistiques d'apprentissage supervisé.

3 Architecture du programme

Dans cette partie nous allons décrire l'implémentation du wrapper en pseudo-code en présentant les différents modules, les fonctions ainsi que les objets générés (voir Figure 1). Le package inclut deux fonctions utilisables par l'utilisateur :

— `wrapper` : retourne le meilleur modèle trouvé avec l'historique des modèles testés

— `unpack` : concatène les valeurs d'un vecteur dans une chaîne de caractères

Les autres fonctions commencent par un point.

Nous avons implémenté les algorithmes backward, forward et génétique, ainsi que les méthodes d'apprentissage suivantes :

— la régression logistique (`logit`)

— l'analyse discriminante linéaire (`lda`) et quadratique (`qda`)

— la régression linéaire (`lm`), avec le critère BIC

— Support Vector Machine (`svm`)

— la classification naïve bayésienne (`naiveBayes`) (Gowda et al., 2010),

— et random forest (`randomForest`)

L'utilisateur peut également utiliser sa propre méthode d'apprentissage avec une fonction au format approprié.

Afin d'évaluer les modèles, nous utilisons le taux d'erreur calculé sur un échantillonnage simple apprentissage/test, que nous cherchons à minimiser (à l'exception du modèle linéaire qui minimise le BIC).

Nous avons évalué le code de notre modèle en utilisant le package microbenchmark. Ce package permet de comparer la vitesse d'exécution de différentes instructions sur plusieurs centaines de mesures. Il donne accès au temps moyen, minimum, maximum, ... Nous ne rentrerons cependant pas plus dans l'utilisation de celui-ci, étant donné que cela relève d'un aspect très technique qui, bien que méritant d'être mentionné, ne trouve pas sa place dans ce rapport.

Le package se découpe en 7 modules (voir Figure 1) :

— `wrapper.R`

— `choose.classifier.R`

— `choose.wrapper.R`

— `obj.builder.R`

— `wrap.back.R`

— `wrap.for.R`

— `wrap.gen.R`

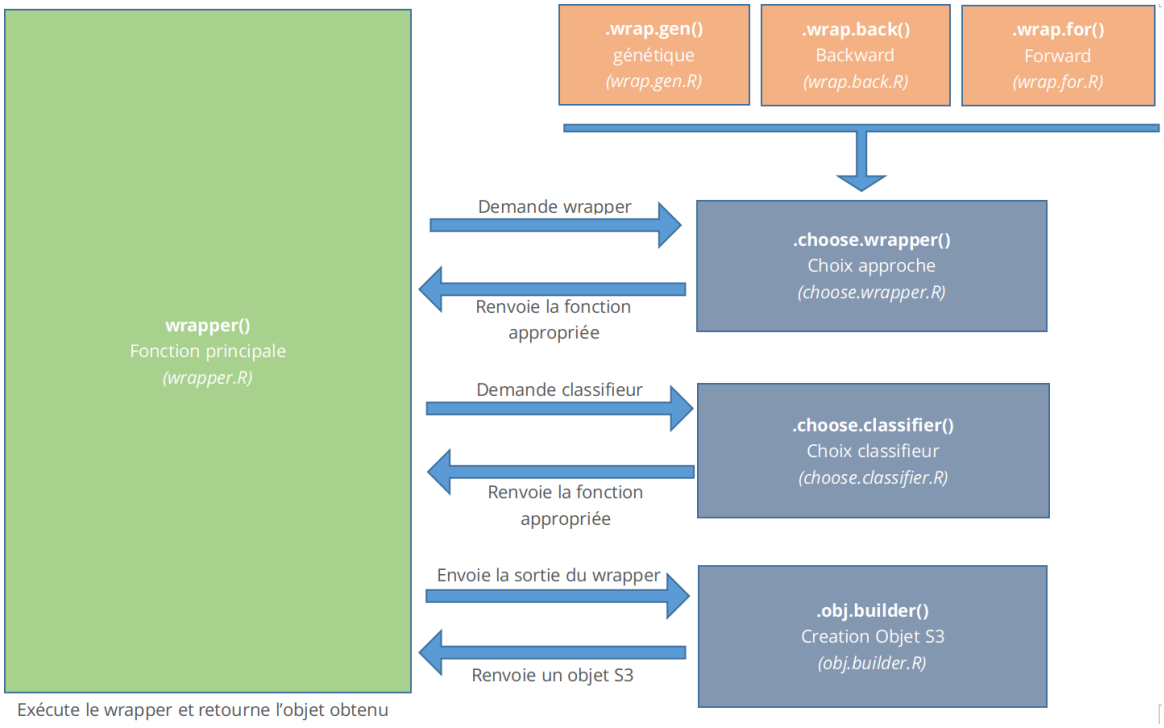

FIGURE 1 – Présentation des différents modules et leurs interactions

Nous détaillons ci-dessous les différents modules du package sous forme de pseudo-code.

3.1 `wrapper.R`

C'est le module principal. Il contient la fonction appelée par l'utilisateur et fait appel aux fonctions des autres modules pour réaliser le wrapper. Seule la fonction mère `wrapper()` y est

incluse (voir Algorithme 1).

input
dataset : jeu de données ;
learner : nom méthode d'apprentissage ;
train : proportion/taille de l'échantillon test (ou directement l'échantillon test) ;
algorithm : nom algorithme de recherche du minimum ;
k.lim : nombre d'itération qui sont réalisées en dehors de l'optimisation du taux d'erreur ;
target : index-colonne/nom de la variable à expliquer du jeu de données ;
features : index-colonne/nom des la variables à explicatives du jeu de données ;
plot.err : graphique représentant le taux d'erreur de tous les modèles testés ;
... : paramètres supplémentaires pour le modèles ;
output
wrapped : un objet contenant l'historique des modèles testés et leur taux d'erreur ainsi que le meilleur modèle ;
begin
Initialisation / Vérification des inputs ;
if *train n'est pas un vecteur* **then**
  if *train est une proportion* **then**
    $train \leftarrow nb.lignes(dataset) * train$;
  end
  $train \leftarrow sousensembledeslignesdedataset, detailletrain$;
end
classifieur ← fonction calculant le taux d'erreur avec la bonne méthode ;
algofun ← fonction de recherche d'optimum choisie ;
resultats ← execution d'algofun ;
resultats ← structuration des resultats ;
if $plot.err = vrai$ **then**
  Affichage de la courbe de l'historique des taux d'erreurs ;
end
return resultat ;
end

Algorithm 1: Fonction `wrapper()`

3.2 `choose.classifier.R`

Ce module recense toutes les méthodes d'apprentissage supervisé que l'utilisateur peut choisir et utiliser sur ses données.

Avant tout, la fonction **unpack** est définie et permet de mettre sous forme de vecteur avec un séparateur toutes les variables explicatives. Cela permet de faciliter le traitement lors de l'exécution du modèle avec (`as.formula`).

La fonction `.choose.classifier` est une fonction cachée qui prend en paramètre le nom de la méthode d'apprentissage souhaitée par l'utilisateur (learner), la variable cible (`target`) ainsi que le jeu de données (`X`). Chaque fonction est construite sous le modèle suivant (voir

Algorithme 2) :

```
if learner = "logit" then
    return fonction (features, ...)
    model ← le modèle est appliqué aux données ;
    prédiction ← la prédiction avec ce modèle est appliquée ;
    tt ← la matrice de confusion est calculée ;
    return le taux d'erreur ;
end
```

Algorithm 2: Exemple de fonction de chaque modèle statistique

Afin de choisir et définir quels packages et méthodes à utiliser dans le package, nous nous sommes inspiré du top 20 des packages R de machine learning du site des KDnuggets [1] ainsi que du top 10 des des algorithmes de machine learning sur le site Data Science Central [2] et DeZyre [3].

Si l'utilisateur préfère plutôt utiliser sa propre fonction d'apprentissage, il peut le faire en indiquant sa fonction plutôt que le nom d'une méthode d'apprentissage dans le paramètre learner.

Sa fonction doit alors suivre le même schéma que montré précédemment (voir Algorithme 3) :

```
fonction (features, ...) # features est le vecteur des noms de features
...
return (value) # value est un indicateur à minimiser ;
```

Algorithm 3: Exemple de fonction de chaque modèle statistique

Un exemple est disponible dans la partie tutoriel.

3.3 `choose.wrapper.R`

Dans ce module, la fonction cachée `.choose.wrapper` permet d'identifier l'approche wrapper que l'utilisateur souhaite appliquer. Elle retourne la fonction du module correspondant à l'approche (`.wrap.for`, `.wrap.back` ou `.wrap.genalg`).

Il y a deux fichiers d'aide pour les fonctions :

— `unpack`, mise en forme du formula

— `wrapper`, fonction principale du wrapper

Les autres fonctions sont internes au package, cachées pour l'utilisateur car compliquées à utiliser. Nous n'avons pas jugé pertinent de les rendre publiques en plus de la fonction wrapper qui offre les mêmes possibilités à l'utilisateur avec une interface simplifiée.

1. Site KDnuggets.
2. Site Data Science Central.
3. Site DeZyre.

3.4 wrap.for.R / wrap.back.R / wrap.gen.R

Tout d'abord, la fonction `wrap.for` permet de sélectionner l'une après l'autre des variables permettant d'optimiser le modèle selon un critère sur le modèle choisi par l'utilisateur. Cette fonction prend en paramètre l'ensembles des variables, le classifieur ainsi que la limite du nombre d'itérations après que le programme ait trouvé un minimum local. Elle retourne le résultat obtenue avec l'approche forward.

La fonction `forward` prend en paramètres les variables utilisées, les variables sélectionnées, le vecteur qui contient les erreurs de tous les modèles testés, le vecteur qui contient les modèles testés, le taux d'erreur du modèle testé ainsi que le nombre d'itérations (le nombre de modèle déjà testé). Voici ci-dessous le code du module `wrap.for` :

```
.wrap.for <- function(features, wrap.learner, k.lim, ...) {
  ## Avoid recursivity limits:
  # we store the option about limit of recursive calls :
  old.exp <- options()["expressions"]
  if (length(features) > old.exp) {
    # we set recursion limits to the number of features :
    options(expressions = length(features))
  }
  forward.search <- function(f, X, ...) {
    # see .wrap.learner.R for more informations
    return(wrap.learner(c(X, f), ...))
  }
  forward <- function(feats.use, feats.set, err.list, mod.list, err.curr, k=0) {
    err.min <- which.min(err.curr)
    if (min(err.list) > err.curr[err.min] && length(feats.set) > 0) {
      ## if no minimum is reach and we have not an empty set of features
      ## we continue recursion
      forward(# update set used features
        feats.use= c(feats.use, feats.set[err.min]),
        # updated set of unused features
        feats.set= feats.set[-err.min],
        # updated error list
        err.list = c(err.list, err.curr[err.min]), #
        # historic of models tested
        mod.list = append(mod.list, list(c(feats.use, feats.set[err.min]))),
        # error rates for the candidates of next step
        err.curr = sapply(feats.set[-err.min], forward.search,
                          c(feats.use, feats.set[err.min]), ...)
      )
    } else {
      ## While there is less than k.lim steps done without a better features
      ## subset
      if (k < k.lim && length(feats.set) > 0) {
        forward(feats.use= c(feats.use, feats.set[err.min]),
                feats.set= feats.set[-err.min],
                err.list = c(err.list, err.curr[err.min]),
                mod.list = append(mod.list, list(c(feats.use, feats.set[err.min]))),
                err.curr = sapply(feats.set[-err.min], forward.search,
                                  c(feats.use, feats.set[err.min]), ...),
                k=k+1
        )
```

```
      } else {
        # All possibilities have been tested or a minimum has been find and
        # unbeaten for k.lim iterations
        hist.feat <- append(mod.list, list(c(feats.use, feats.set[err.min])))
        hist.err  <- c(err.list, err.curr[err.min])[-1]
        return(
          list(best.feats= hist.feat[[which.min(hist.err)]],
               hist.feat = hist.feat,
               hist.err  = unlist(hist.err)
          )
        )
      }
    }
  }
   # Launch the recursivity with no used features
  result <- forward(c(), features, Inf, c(),
                    sapply(features, forward.search, c(), ...))

  # we go back to the previous settings for recursivity
  if (length(features) > old.exp) options(expressions=old.exp)
  return(result)
}
```

Le premier bloc de cette fonction permet de rechercher le taux d'erreur minimum. Si aucun minimum n'est trouvé et qu'il n'y a pas de variables sélectionnées alors la fonction forward recommence en prenant d'autres valeurs de paramètres comme l'historique des modèles testés. Cet historique est établi à chaque nouvelle itération, le modèle testé ainsi que les taux d'erreur obtenus sont stockés dans un vecteur.

Sinon, lorsqu'un minimum est trouvé, le second permet bloc de vérifier que la limite des itérations n'est pas atteinte et qu'il y a au moins une variable sélectionnée. Tant que cette condition est remplie, les paramètres de la fonction forward sont actualisés.

Lorsque toutes les combinaisons linéaires de variables ont été testées, le dernier bloc permet de récupérer les meilleures variables qui minimisent le taux d'erreur du modèle. Le sous ensemble de variables est retournée par la fonction wrap.for lorsque le nombre de variables sélectionnées est supérieur aux sélections précédentes, sinon un message d'erreur est retourné.

Ensuite, la fonction `wrap.back` fonctionne globalement avec le même principe que la fonction wrap.for. Les noms des paramètres en entrée sont identiques mais ils ne prennent pas les mêmes valeurs. En effet, la fonction commence ses itérations avec toutes les variables explicatives. afin de les retirer une à une tant que cela permet l'amélioration du modèle (diminution du taux d'erreur). Voici ci-dessous le code du module `wrap.back`.

```
.wrap.back <- function(features, wrap.learner, k.lim, ...) {
  ## Avoid recursivity limits:
  # we store the option about limit of recursive calls :
  old.exp <- options()["expressions"]
  if (length(features) > old.exp) {
```

```
  # we set recursion limits to the number of features :
  options(expressions = length(features))
}
back.search <- function(f, X, ...) {
  # see .wrap.learner.R for more informations
  return(wrap.learner(X[X!=f], ...))
}
backward <- function(feats.use, err.list, mod.list, err.curr, k=0) {
  err.min <- which.min(err.curr)
  if (min(err.list) > err.curr[err.min] &&
      length(feats.use) > 2) {
    ## if no minimum is reach and we have not an empty set of features
    ## we continue recursion
    feats.use <- feats.use[-err.min]
    backward(# update set used features
      feats.use= feats.use,
      # updated error list
      err.list = c(err.list, err.curr[err.min]),
      # historic of models tested
      mod.list = append(mod.list, list(feats.use)),
      # error rates for the candidates of next step
      err.curr = sapply(feats.use, back.search, feats.use, ...)
    )
  } else {
    ## While there is less than k.lim steps done whithout a better features
    ## subset
    if (k < k.lim && length(feats.use) > 2) {
      feats.use <- feats.use[-err.min]
      backward(feats.use= feats.use,
               err.list = c(err.list, err.curr[err.min]),
               mod.list = append(mod.list, list(feats.use)),
               err.curr = sapply(feats.use, back.search, feats.use, ...),
               k=k+1
      )
    } else {
      # All possibilities have been tested or a minimum has been find and
      # unbeaten for k.lim iterations
      hist.feat <- append(mod.list, list(feats.use[-err.min]))
      hist.err  <- c(err.list, err.curr[err.min])
      return(
        list(best.feats= hist.feat[[which.min(hist.err)]],
             hist.feat = hist.feat,
             hist.err  = hist.err
        )
      )
    }
  }
}
# Launch the recursivity with no used features
result <- backward(features, wrap.learner(features),
                   list(features), sapply(features, back.search, features, ...))
# we go back to the previous settings for recursivity
if (length(features) > old.exp) options(expressions=old.exp)
return(result)
}
```

Tant qu'il reste au moins 2 variables explicatives et que la limite fixée par l'utilisateur n'est

pas atteinte, l'algorithme continue ses itérations afin de retirer progressivement des variables de l'analyse qui maximisent le taux d'erreur.

Enfin, la fonction **wrap.gen** prend en paramètre l'ensemble des variables explicatives des données (**features**), la méthode d'apprentissage souhaitée par l'utilisateur ainsi que le nombre d'itérations qui limite la recherche d'un minimum local (**k.lim**) durant la recherche de solutions optimales même après que le wrapper ait trouvé un minimum local. Cette fonction peut aussi prendre des paramètres supplémentaires mentionnés par l'utilisateur. Le seuil du niveau d'erreur est de 30% (voir Algorithme 4).

fonction wrap.gen (features, wrap.learner, k.lim, ...) ← application de la méthode d'apprentissage aux variables explicatives et retour des erreurs (résidus du modèle) ;
fonction evalFunc (x)
x est un vecteur binaire : 1 si la variable est sélectionnée, 0 sinon comparaison entre l'erreur du modèle en cours par rapport au seuil d'optimisation ;
err.curr ← taux d'erreur sur l'ensemble des features i telles que $x[i] == 1$;
if *err.curr* > 0.3 **then**
  return 1 ;
else
  return err.curr # résultat à retourner en fonction du résultat de la comparaison ;
end
fin fonction
GAmodele ← fonction de l'algorithme génétique appliquée au modèle ;
res.curr ← récupération du taux d'erreur de GAmodele et des variables choisises ;
return liste des variables sélectionnées ;
fin fonction
fin fonction

Algorithm 4: Algorithme génétique

NB : l'algorithme génétique ne gère pas les paramètres supplémentaires pour la méthode d'apprentissage.

3.5 `obj.builder.R`

Ce module est chargé de la création de l'objet retourné par `wrapper()`. Il contient deux fonctions. La première, `.obj.builder()`, prend en input les paramètres utilisés pour l'exécution du wrapper ainsi que ses résultats. Elle retourne un objet `S3` de classe "wrapper", qui est en fait une liste contenant :

- `algorithm` : le nom de l'algorithme wrapper utilisé
- `classifier` : le nom de la méthode d'apprentissage
- `formula` : la formule permettant de retrouver le modèle (objet de type "language", obtenu avec as.formula)
- `best.feats` : la liste des meilleurs features
- `hist.mods` : l'historique des sous-ensembles de features testés (sous forme de liste)
- `hist.errors` : vecteur des taux d'erreurs des modèles testés

— `error.rate` : le taux d'erreur du meilleur modèle
— `train.sample` : vecteur des numéros des lignes utilisées pour l'apprentissage
— `para.sup` : liste de paramètres supplémentaires

La deuxième fonction, `print.wrapper()`, est en fait la méthode de la classe "wrapper" appelée lors de l'utilisation des fonctions `show()` et `print()` sur un objet de type "wrapper". Elle permet de mettre en forme l'affichage de notre objet. Le formatage que nous avons choisi est le suivant :

```
# wrapper : [nom du wrapper] #
Call :
[instruction R complète permettant de retrouver le meilleur modèle]
error.rate: [taux d'erreur]
precision: [precision]
tested models: [nombre de modèles testés]
```

4 Tutoriel

4.1 Installation

Afin d'installer le package `wrap` et de disposer de toutes ses fonctionnalités, il est conseillé d'installer préalablement les packages "`randomForest`", "`MASS`", "`e1071`" et "`genalg`". L'importation de l'archive du package `wrap.zip` peut se faire depuis le menu `Tools/Install Packages...` de RStudio.

```
filepath <- "/Chemin/vers/wrap.zip"
install.packages("randomForest")
install.packages("e1071")
install.packages("genalg"))
install.packages("MASS")
install.packages(filepath, repos = NULL, type = "win.binary")
```

4.2 Wrapper simple

```
library("wrap")
## 1. Wrapper simple sur régression linéaire --------------------------- #
## On cherche à prédire la quantité maximale d'O3 d'après les autres features
data(ozone)  # Chargement du jeu de données ozone
head(ozone)  # Visualisation des premières valeurs
summary(ozone)

# La variable VENT12 est qualitative, on la retire du dataset
ozone <- ozone[, -8]
# maxO3 la target est en dernière position, et on veut choisir nos features parmi
# toutes les autres variables. On a donc pas besoin de renseigner les champs
# features et target.
wrapped <- wrapper(dataset = ozone,
                   learner = "lm"
)
print(wrapped)
# Le meilleur modèle selon le critère BIC
model <- lm(wrapped\$formula, data=ozone)
```

4.3 Paramètres supplémentaires pour le classifieur

```
# On peut ajouter des paramètres pour le classifieur
# par exemple on peut faire plusieurs svm avec des kernels different et voir
# lequel est le plus efficace
data(diabete)
head(diabete)
# on ne veut pas prendre la variable DiabetesPedigreeFunction car c'est un factor
# SVM radial
wrapped <- wrapper(dataset = diabete,
                   learner = "svm",
                   algorithm = "forward",
                   features = (1:8)[-7],
                   kernel = "radial"
)
# SVM Lineaire
wrapped <- wrapper(dataset = diabete,
                   learner = "svm",
                   algorithm = "forward",
                   features = (1:8)[-7],
                   kernel = "linear"
)
# SVM Polynomial
wrapped <- wrapper(dataset = diabete,
                   learner = "svm",
                   algorithm = "forward",
                   features = (1:8)[-7],
                   kernel = "polynomial"
)
```

4.4 Wrapper sur crédits avec échantillon d'apprentissage fixé

```
data(income)
head(income)

train.samp <- sample(1:nrow(income), size = 3500)
head(train.samp)  # indices des lignes prises en apprentissage

# on peut comparer des methodes differentes
wrapped.nb <- wrapper(dataset = income,
                   learner = "naiveBayes",
                   algorithm = "forward",
                   features = c(1, 3, 5, 11, 12, 13),
                   k.lim=1,  # on veut tester un minimum de modele pour
                   # reduire le temps d'execution
                   train = train.samp
)
wrapped.svm <- wrapper(dataset = income,
                       learner = "svm",
                       algorithm = "forward",
                       features = c(1, 3, 5, 11, 12, 13),
                       k.lim=1,
                       train = train.samp
)
# Comparons les résultats
print(paste("Naive  |  error:", wrapped.nb$error.rate,
            "nb.feats:", length(wrapped.nb$best.feats)))
```

```
print(paste("SVM    |  error:", wrapped.svm$error.rate,
            "nb.feats:", length(wrapped.svm$best.feats)))
```

4.5 Wrapper sur Income avec notre propre fonction d'évaluation (cross-validation)

```
# Utilisons notre propre fonction pour prédire
# le revenu en utilisant une 10-folds validation
# et le classifieur naif bayesien

# Voyons comment intégrer notre propre fonction d'évaluation
evalFUN <- function(features, ...) {
# la fonction ne doit pas avoir d'autres parametres
# toutes les autres variables doivent être connu hors de la fonction
  k_folds <- 10
  folds_i <- sample(rep(1:k_folds, length.out = nrow(income)))
  # Une validation
  error_rate <- function(k) {
    test_i <- which(folds_i == k)
    model <- e1071::naiveBayes(as.formula(paste("income~", unpack(features))),
                           data=income[-test_i, ])
    prediction <- predict(model, income[test_i, ])
    tt <- table(income[test_i, "income"], prediction)
    return(sum(tt[row(tt) != col(tt)]) / sum(tt))
    # retourne le taux d'erreur (à minimiser)
    # si vous voulez utiliser un indicateur
    # qui doit etre maximisé, multipliez le par -1
  }
  return(mean(sapply(1:k_folds, error_rate))) # moyenne sur toute les validations
}

# les autres en features
wrapped <- wrapper(dataset = income,
                   learner = evalFUN,
                   algorithm = "forward"
)
```

5 Conclusion

Le package wrap que nous avons créé au cours de ce projet permet de sélectionner les features les plus efficaces. Notre wrapper minimise le taux d'erreur du modèle pour évaluer les sous-ensembles de features. Il propose trois stratégies de sélections : forward, backward et algorithme génétique.

Il offre également un choix de méthodes d'apprentissage varié, avec la possibilité d'ajouter des paramètres supplémentaires :

- la régression logistique
- l'analyse discriminante (linéaire et quadratique)
- la régression linéaire avec le critère BIC
- Support Vector Machine

— la classification naïve bayésienne
— random forest

De plus, l'utilisateur peut utiliser sa propre fonction et donc le modèle et l'évaluation de son choix (par exemple maximiser le F-Score avec la cross-validation).

Le package retourne une liste avec un affichage personnalisé et peut afficher la courbe des taux d'erreur (ou autres mesures utilisées).

Nous avons optimisé au maximum le code pour qu'il s'exécute rapidement et pour qu'il minimise les contraintes d'application de la fonction par rapport à la taille du jeu de données de l'utilisateur.

Certaines améliorations pourraient être faites. Tout d'abord dans le but de gagner encore en performances, certaines fonctions pourraient être implémentées en C, langage bien plus rapide que les scripts R. De plus, le package n'offre pas d'options pour choisir une méthode d'évaluation particulière. Nous pourrions donc, pour aller plus loin, rajouter une implémentation de bootstrap et de la cross-validation.

Aussi, ce travail nous a permis d'assimiler des concepts théoriques autant en programmation sous R (packages, optimisation du code, ...) que sur les méthodes statistique d'apprentissage supervisé et la sélection de features. Nous avons pu réaliser un package qui peut être diffusé et installé de façon autonome et qui comprend une documentation sur les fonctions `wrapper()` et `unpack`.

Globalement, ce projet nous a permis de mettre en application les notions théoriques abordées en cours, mais aussi de découvrir la création de package R malgré les difficultés rencontrées comme le fait de s'approprier le concept de wrapper ainsi que les différentes méthodes statistiques et de trouver une architecture optimale du programme.

Ce projet a été pour nous enrichissant et nous permettra, à l'avenir, de valoriser cette compétence.

6 Bibliographie

Gowda, K. A., M.A., J., and A.S., M. (2010). Feature subset selection problem using wrapper approach in supervised learning. *International Journal of Computer Applications*, 1(7) :0975 – 8887.

Grzegorz, D. (2010). Tournament searching method to feature selection problem. pages 42 – 200.

Hall, M. A. and Holmes, G. (2003). Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions on Knowledge and Data Engineering*, 15(6) :1437–1447.

Jonathan, R. S., Peter, N., John, C., Jitendra, M., and Douglas, E. (2003). *Artificial intelligence : a modern approach*, volume 2. Prentice hall Upper Saddle River.

Rebaïne, D. (2005). Note de cours : La méthode de branch and bound. Technical report, Université du Québec à Chicoutimi.

Ron, K. and George, H. (1997). Wrappers for feature subset selection. *Elsevier*, pages 273–324.