ofw: an R package to select continuous variables for multiclass classification with a stochastic wrapper method
Kim-Anh Lê Cao, Patrick Chabrier

To cite this version:
Kim-Anh Lê Cao, Patrick Chabrier. ofw: an R package to select continuous variables for multiclass classification with a stochastic wrapper method. 2008. <hal-00300209>

HAL Id: hal-00300209
https://hal.archives-ouvertes.fr/hal-00300209
Submitted on 17 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
ofw: an R package to select continuous variables for multiclass classification with a stochastic wrapper method

Kim-Anh Lê Cao¹,² and Patrick Chabrier³

Abstract

When dealing with high dimensional and low sample size data, feature selection is often needed to help reduce the dimension of the variable space while optimizing the classification task. Few tools exist for selecting variables in such data sets, especially when classes are numerous (> 2).

We have developed ofw, an R package that implements, in the context of classification, the meta algorithm “Optimal Feature Weighting” (OFW). We focus on microarray data, although the method can be applied to any $p >> n$ problems with continuous variables.

The aim is to select relevant variables and to numerically evaluate the resulting variable selection. Two versions of OFW are proposed with the application of supervised multiclass classifiers such as CART and SVM. Furthermore, a weighted approach can be chosen to deal with unbalanced multiclass, a common characteristic in microarray data sets.

ofw is freely available as an R package under the GPL license. The package can be downloaded from the Comprehensive R Archive Network (CRAN).

Introduction

Performing a feature selection algorithm has several important applications in high dimensional data sets. For example with microarray data, it is sensible to use a dimensional reduction technique, either to identify genes that contribute the most for the biological outcome (e.g. cancerous vs. normal cells) and to determine in which way they interact to determine the outcome, or to predict the outcome when a new observation is presented. Such a method would provide practical aspects with machine learning methods: it avoids the “curse of dimensionality” that leads to overfitting when the number of variables is too large.

There are two ways of selecting features. Either explicitly (filter methods) or implicitly (wrapper methods). The filter methods measure the relevance of a feature at a time by performing statistical tests (e.g. t-, F-tests) and ordering the p-values. This type of approach is robust against overfitting and is fast to compute. However, it usually disregards the interactions

¹Institut de Mathématiques, Université de Toulouse et CNRS (UMR 5219), F-31062 Toulouse, France
²Station d’Amélioration Génétique des Animaux UR 631, Institut National de la Recherche Agronomique, F-31326 Castanet, France
³Biométrie et Intelligence Artificielle, UR875, Institut National de la Recherche Agronomique, F-31326 Castanet, France
between the features as it tests one variable at a time. Chen et al. (2003) compared four filter methods and reached this conclusion.

The wrapper methods measure the usefulness of a feature subset by searching the space of all possible feature subsets. The search can be performed either with heuristic or stochastic search. The main disadvantages of these methods are their tendency to overfit and when dealing with numerous variables, an exhaustive search is computationally impossible. However, the resulting selection takes into account the interactions between variables and might highlight useful information on the experiment. Despite this latter property, wrapper methods are still not widely applied in microarray data. Comparisons of Random Forests (Breiman, 2001), Recursive Feature Elimination (Guyon et al., 2002), $L_0$ norm SVM (Weston et al., 2003) and biological interpretation of the resulting gene selections is given in Lê Cao et al. (2007b).

In this R package, we implement the wrapper method “Optimal Feature Weighting” (OFW) adapted from Gadat and Younes (2007) that numerically quantifies the classification efficiency of each variable with a probability weight, by using stochastic approximations. This meta algorithm can be applied to any classifier. Therefore, the classifiers SVM (Support Vector Machines, Vapnik (1999)) and CART (Classification and Regression Trees, Breiman et al. (1984)) have been implemented so as to select an optimal subset of discriminative variables. Few wrapper methods have been proposed yet to deal with multiclass data sets (Li et al., 2004; Chen et al., 2003; Yeung and Burmgarner, 2003), especially when the classes are unbalanced (Chen et al., 2004). Our function `ofw()` proposes a weighting approach to deal with this common characteristic in microarrays.

Furthermore, like any wrapper methods, `ofw()` requires heavy computations, especially when the number of variables is large. In this package, some of the computation time has been reduced by implementing some C functions and by proposing parallel programming during the learning step.

Finally, we propose to perform the $e.632+$ bootstrap method (Efron and Tibshirani, 1997) to estimate the classification error rate on bootstrap samples and to evaluate the different variants of OFW and the resulting gene selections.

The general principle of the OFW algorithm is first presented. We then detail how to use `ofw()` by applying the main functions on one microarray data set that is available in the package.

1 Optimal Feature Weighting model

Principle

Optimal Feature Weighting (OFW, Gadat and Younes 2007) is a meta algorithm that can treat several classification problems with a feature selection task. Any classifier can be applied, and Lê Cao et al. (2007a) implemented OFW with CART and SVM for multiclass classification (see also Lê Cao et al. 2007b for binary case).

We assume that the $n$ examples (or cases) are described by $p$ attributes (or variables) and labelled with their target class (e.g. $\{0, 1\}$ in binary problems).

Given a probability weight vector $\mathbb{P}$ on all $p$ variables, the key idea of OFW is to learn $\mathbb{P}$ such that it fits the classification efficiency of each variable in the given problem. In short, important weights will be given to variables with a high discriminative power, and low or zero weights to non relevant variables in the classification task.

For that purpose, the algorithm adopts a wrapper technique, by drawing a small variable
subset $\omega$ at a time, by measuring the relevance of this subset with the computation of the classification error rate, and then by updating the probability weights $P$ according to the discriminative power of the variable subset $\omega$. As an exhaustive search of the whole variable space is not tractable when $p$ is large (in microarray data $p > 5000$), stochastic approximations are proposed, see Gadat and Younes (2007); Lê Cao et al. (2007b) for the detailed theory of the model. At iteration $i$ in the algorithm, the probability weight vector is updated with a gradient descent:

$$P_{i+1} = \Pi_S [P_i - \alpha_i d_i]$$

where $\Pi_S$ is the projection on the simplex of probability map on the set of variables, so that $P_{i+1}$ remains a probability vector, $\alpha_i$ is the step of the gradient, and $d_i$ is the stochastic approximation of the gradient (see below).

The whole process is repeated $\text{iter.max}$ times and the final output is $P_{\text{iter.max}}$, that indicates the importance of each variable in the data. To obtain a variable selection, the user only needs to rank the variables according to their decreasing weights, and to choose the length of the selection.

### General algorithm

**Input:** a data matrix of size $n \times p$ and the class values vector of size $n$

**Parameters:** number of total iterations $\text{iter.max}$ and the size $\text{mtry}$ of the variable subset $\omega$

**Output:** $P_{\text{iter.max}}$ a weight vector of length $p$

**Initialize** $P_0 = [1/p \ldots 1/p]$ (uniform distribution on all variables)

**For** $i = 1$ to $\text{iter.max}$

1. Variables: draw a subset $\omega_i$ with respect to $P_i$
2. Cases: draw a bootstrap sample $B_i$ in $1 \ldots n$, define $\tilde{B}_i$ the out-of-bag cases
3. Train the classifier on variables in $\omega_i$ and cases in $B_i$
4. Test the classifier on variables in $\omega_i$ and cases in $\tilde{B}_i$, compute the classification error rate $\epsilon_i$
5. Compute the drift vector $d_i$
6. Update $P_{i+1} = \Pi_S [P_i - \alpha_i d_i]$

where:

- $d_i = \frac{C(\omega_i,k)\epsilon_i}{\sum_{j} C(\omega_i,j)}$ is the approximated gradient, and $C(\omega_i,k)$ is the number of occurrences of variable $k$ in the subset $\omega_i$, in case this variable is drawn more than one time in $\omega_i$.
- $\Pi_S$ is the projection on the simplex, so that $\sum_{j} P_{ij} = 1$ and $\forall i \quad P_{ii} \geq 0$, $i = 1 \ldots p$.
- $\alpha_i$ is the step of the gradient descent, and can be set to $\frac{1}{\epsilon + 1}$.

1.1 **OYW is applied with either CART or SVM**

We applied OFW with two supervised algorithms: Support Vector Machines (SVM) and Classification And Regression Trees (CART).
Support Vector Machines

SVM SVM (Vapnik, 1999) is a binary classifier that attempts to separate the cases by defining an optimal hyperplane between the 2 classes up to a consistency criterion. Linear kernel SVMs are performed here because of their good generalization ability compared to more complex kernels.

SVM for multiclass data We applied OFW with the one-vs.-one SVM approach that is implemented in the e1071 R package. OFW hence depends on e1071. The user only needs to set the total number of iterations to perform (nsvm) and the size mtry of the subset ω to draw at each iteration (see section 3.2 for tuning).

Classification And Regression Trees

OFW is applied with the multiclass classifier CART (Classification And Regression Trees, Breiman et al. 1984) that is well adequate for multiclass problems. Following the example of Breiman (1996), the trees were aggregated (bagging) to overcome their unstable characteristic. Hence, several classification trees are constructed on different bootstrap samples and with different subsets ω. The approximated gradient is also slightly modified. The modified algorithm is as follows:

Input: data matrix of size n × p and the class values vector of size n
Parameters: number of total iterations iter.max, the size mtry of the variable subset ω and the number ntree of trees to aggregate
Output: P_{iter.max} a weight vector of length p

Initialize P_0 = [1/p \ldots 1/p] (uniform distribution on all variables)
For i = 1 to iter.max
  1. For b = 1 to ntree
     (a) Variables: draw a subset ω^b_i with respect to P_i
     (b) Cases: draw a bootstrap sample B^b_i in 1 \ldots n, define |B^b_i| the out-of-bag cases
     (c) Train the classifier on variables in ω^b_i and cases in B^b_i
     (d) Test the classifier on variables in ω^b_i and cases in |B^b_i|, compute the classification error rate \epsilon^b_i
  2. Compute the drift vector D_i
  3. Update P_{i+1} = \Pi_S[P_i - \alpha_i D_i]

where D_i is an averaged time version of the gradient d_i (see Lê Cao et al., 2007b).
Hence, as in Random Forests (Breiman 2001), ntree trees are constructed on ntree bootstrap samples. The only difference lies in the construction of the classification trees: instead of randomly selecting a variable subset to split each node of each tree (Random Forests), the variable subset is drawn with respect to the probability P_i to construct each tree.
In addition to choose the total number of iterations to perform (nforest) and the size mtry
of the subset $\omega$ to draw at each iteration, the user needs to choose the number of aggregated
trees $n_{\text{tree}}$ (see section 3.2 for tuning).

1.2 Unbalanced Multiclass

**Challenge when data are unbalanced**

Multiclass problems are often considered as an extension of 2-class problems. However this
extension is not always straightforward, especially in microarray data context. Indeed, the
data sets are often characterized by unbalanced classes with a small number of cases in at
least one of the classes. This imbalance is often due to rare classes (e.g. a rare disease
where patients are few) that are biologically interesting. Nevertheless, most algorithms do
not perform well for such problems as they aim to minimize the overall error rate instead of
focusing on the minority class.

**Weighted procedure in OFW: wOFW**

An efficient way to take into account the unbalanced characteristics of the data set is to weight
the error rate $\epsilon_i$ according to the number samples of each class in the bootstrap sample. This
allows for penalizing a classification error made on the minority class and, therefore, put more
weight on the variables that help classify this latter class instead of the majority one (Lê Cao
et al., 2007a).

This weighted approach has been implemented in both versions of the algorithm, called ofw-
CART and ofwSVM, and also stands for the evaluation step (step 2 in Fig. 1 and see section
3.4).

2 Implementation issues

**ofw** is available from the Comprehensive R Archive Network (CRAN\footnote{http://CRAN.R-project.org/}, or one of its mirrors). Instructions for package installation are given by typing \texttt{help(INSTALL)} or \texttt{help(install.packages)}
in R.

**ofw** is a set of R and C functions to perform either ofwCART or ofwSVM and to evaluate
the performances of both algorithms. Two classes of functions in R and C are implemented.
Figure 1 provides a schematic view of the analysis of a data set with ofw. Each step in Fig.
1 will be detailed in section 3 on a small microarray data set.

The R environment is the only user interface. The R procedure calls a C subroutine, whose
results are returned to R. There is no formula interface and the predictors can be specified as
a matrix or a data frame via the \texttt{x} argument, with factor responses as a vector via the \texttt{y} argument. Note that ofw performs only classification and does not handle categorical variables.

Details of the components of each object from ofw\texttt{Tune}, ofw, learn, evaluate and evaluate
CARTparallel are provided in the online documentation. Methods provided for the classes
ofw\texttt{Tune} and ofw include \texttt{print}.

The C function \texttt{classTree.c} that constructs classification trees has been borrowed from
the Breiman and Cutler’s Fortran programs and converted to C language. The function `agregTree.c` that aggregates trees was then largely inspired from the `randomForest` package (Liaw and Wiener, 2002).

3 Using ofw

We detail the call to functions and R commands (preceded by the prompt symbol `>` of `ofw`, that can be loaded into R by `> library(ofw)`.

3.1 Illustrative data set

`ofw` was previously tested on several published microarray data sets (Lê Cao et al., 2007b,a) by comparing it with several other wrapper algorithms. We comment on the present paper the results obtained on one data set that is provided as an example in the package.

SRBCT (Khan et al., 2001) is the data set of small round blue cell tumors of childhood. The training set consists of 63 training samples spanning 4 classes. The data set available in the package includes 2308 genes out of the 6567 after filtering for a minimal level of expression (performed by Khan et al., 2001). Further details about this data set can be found in [http://research.nhgri.nih.gov/microarray/Supplement](http://research.nhgri.nih.gov/microarray/Supplement).

In order to minimize the computation time in this illustrative example, we have reduced SRBCT to 200 genes by simply randomly selecting these out of the 2308 in the initial data set. We also added a factor `class` that indicates the class of each microarray sample.
Note that normalization of the data, that is a crucial step in the analysis of microarray data is not dealt with \textit{ofw} and has to be performed first by the user.

### 3.2 Tuning parameters

In the algorithm OFW, there are mainly 2 to 3 parameters to tune according to the applied classifier to ensure that OFW converges (\textit{step 1} in Fig. [1]):

1. the size of the gene subset $\omega$ (called \texttt{mtry}).
2. the total number of iterations (called \texttt{nsvm} for \texttt{ofwSVM} and \texttt{nforest} for \texttt{ofwCART}).
3. the number of trees \texttt{ntree} to aggregate for \texttt{ofwCART}.

The package \texttt{ofw} provides the function \texttt{ofwTune} to tune these parameters. Here is the command to launch \texttt{ofwTune} with \texttt{ofwCART} for different \texttt{mtry} values:

```r
> data(srbct)
> attach(srbct)
> tune.cart <- ofwTune(srbct, as.factor(class), type="CART", ntree=150,
+ nforest=3000, mtry.test=seq(5,25,length=5), do.trace=100, nstable=25)
> detach(srbct)
```

Note that the only arbitrary parameter that is not tuned and has to be provided by the user is the number of variables \texttt{nstable} one wants to select (see below).

#### Tuning \texttt{mtry}.

The function \texttt{ofwTune} consists in testing OFW (with CART or SVM) with several sizes of the subset $\omega$ (\texttt{mtry.test}). Then, for each \texttt{mtry.test}, OFW is performed twice, called \texttt{ofw1} and \texttt{ofw2}. The first \texttt{nstable} variables with the highest weights in $P_{\text{ofw1}}$ and $P_{\text{ofw2}}$ are extracted. The \texttt{ofwTune} function then outputs the intersection length of these two variable selections.

For example, to tune the parameters with \texttt{ofwCART}:

```r
> tune.cart$param
  1 2 3 4 5
mtry  5 10 15 20 25
length 13 9 9 7 5
```

This outputs the intersection length of the first \texttt{nstable} variables for each tested \texttt{mtry.test}. The value \texttt{mtry=5} gets the best stable results and should be chosen for \texttt{steps 2} and \texttt{3} in Fig. [1] (evaluation and variable selection steps).

#### Early stopping.

Instead of running OFW for all iterations, the user can choose instead to set the number of variables (\texttt{nstable}) to select in the final variable selection step (\texttt{step 3}). This halts the algorithm once it becomes “stable”, that is, when the \texttt{nstable} features of highest weights in $P_i$ and $P_{i+\text{do.trace}}$ are the same for iterations $i$ and $i + \text{do.trace}$.

Finally, to choose the total number of iterations in \texttt{step 3}, we simply suggest to take 2 to
3 times the number of iterations that were performed using the early stopping criterion, to ensure the convergence of the algorithm. This command outputs the number of iterations which were performed:

> tune.cart$itermax
     1 2 3 4 5
ofwCART1 700 500 900 100 100
ofwCART2 800 700 500 800 800

Here the two algorithms ofwCART1 and ofwCART2 stopped at 700 and 800 iterations for mtry=5. During the final learning step, the user should hence set nforest = 3 * 800.

**Tuning ntree (ofwCART).** The best way to tune ntree would be then to run ofwTune with different values of ntree and choose the one that gets the largest intersection length of the first nstable variables. In our experience, the more numerous the trees, the more stable the results, usually for ntree=100 to 150. The same stands for the weighted (weight=T) or non-weighted (weight=F) versions of OFW.

**An example with ofwSVM.** With the SVM classifier, the user has to specify type="SVM" and use nsvm instead of nforest to indicate the number of chosen iterations. As SVM are not aggregated, the user should set nsvm >> nforest.

> tune.svm <- ofwTune(data, as.factor(class), type="SVM", nsvm=200000, mtry=5,  
+ mtry.test=seq(5,25,length=5), do.trace=2000, nstable=25)

> tune.svm$param
     1 2 3 4 5
mtry 5 10 15 20 25
length 7 6 6 1 2

> tune.svm$itermax
     1 2 3 4 5
ofwSVM1 8000 4000 8000 4000 4000
ofwSVM2 10000 10000 12000 4000 10000

In this case, with ofwSVM, the user should set mtry= 5 and nsvm= 30000 for the learning step if nstable=25.

For both classifiers, we strongly advise to choose the smallest mtry that gives the more stable results. Our experience shows that for ofwCART, mtry will be rather small (5 to 15), as the trees are aggregated. For ofwSVM, mtry will usually be larger (> 15). In both cases, mtry should not be greater than nstable, and, therefore, mtryTest ≤ nstable.

Table I illustrates the tuned parameters for several public data sets that were tested in Lè Cao et al. (2007b) and Lè Cao et al. (2007a) for the weighted and non weighted versions of OFW.
Table 1: Values of the size of the subset ω.

<table>
<thead>
<tr>
<th></th>
<th>#genes</th>
<th>#classes</th>
<th>#obs.</th>
<th>ofwCART</th>
<th>w-ofwCART</th>
<th>ofwSVM</th>
<th>w-ofwSVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lymphoma</td>
<td>4026</td>
<td>3</td>
<td>62</td>
<td>5(^1)</td>
<td>10(^1)</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Leukemia</td>
<td>3000</td>
<td>3</td>
<td>72</td>
<td>5(^1)</td>
<td>5(^1)</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>SRBCT</td>
<td>2308</td>
<td>4</td>
<td>63</td>
<td>5(^1)</td>
<td>10(^1)</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Brain</td>
<td>1963</td>
<td>5</td>
<td>42</td>
<td>5(^1)</td>
<td>25(^1)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Follicle</td>
<td>1564</td>
<td>3</td>
<td>42</td>
<td>10(^2)</td>
<td>10(^2)</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

The number of trees aggregated is \(^1\)ntree = 150 and \(^2\)ntree = 100.

3.3 Variable selection and visualization plots

Once the parameters \(mtry\) and \(ntree\) for ofwCART, have been chosen, the variable selection step (step 3 Fig. 1) can be performed, preferably on the whole data set if the sample size is too small, i.e. if \(n\) is roughly less than 80, or if the number of observations per class is too small. We advise to use the total number of iterations \(nforest\) or \(nsvm\), rather than the \(nstable\) early stopping criterion to halt the algorithm, as suggested in section 3.2.

The classifier to be applied has to be specified by the user. Here is the command for the variable selection step (step 3) for ofwCART and ofwSVM.

```r
> learn.cart <- ofw(srbct, as.factor(class), type="CART", ntree=150, + nforest=2500, mtry=5)
> learn.svm <- ofw(srbct, as.factor(class), type="SVM", nsvm=30000, mtry=5)
```

In the case of ofwCART, the evolution of the internal mean error rate \(\bar{\epsilon}_i = \frac{1}{ntree} \sum_{k=1}^{ntree} \epsilon^k_i\) can be plotted for each iteration \(i\), as shown in Figure 2 for \(i = 1 \ldots nforest:\)

```r
> plot(learn.cart$mean.error, type="l")
```

The monotonic decreasing trend of \(\bar{\epsilon}_i\) indicates if the parameters have been tuned correctly and thus if ofwCART converges. In the case of ofwSVM, the SVM are not aggregated, and the error variance is consequently very large: no decreasing trend can be observed and \(\epsilon_i\) is not provided.

Note that the internal error \(\bar{\epsilon}_i\) does not evaluate the performance of OFW (see below section 3.4) and is simply a way to assess the quality of the tuning.

One can also visualize the probability weights \(P_{nforest}\) or \(P_{nsvm}\) for each variable (Figures 3 (a) and (b)):

```r
> plot(learn.cart$prob, type="h")
> plot(learn.svm$prob, type="h")
```

The selected variables can then be extracted by sorting the heaviest weights in \(P\), here for example for the 10 most discriminative variables:
As $P$ is a weight probability, the more numerous the variables, the smallest the weights on the variables. Hence, these weights are a qualitative rather than a quantitative importance measure of the variables, and the choice of a threshold is not advised. The different computations of the approximated gradient in ofwSVM ($d_i$) and in ofwCART ($D_i$), where $D_i \gg d_i$, actually lead to an important number of weights in $P$ close to zero in ofwCART. Remark that some of the very discriminative variables get important weights in both methods, but usually, as the classifiers SVM and CART are differently constructed, the resulting variable selections will not be the same.

### 3.4 Evaluation step

**Method and implementation**

To assess the performance of the variable selection performed by OFW (step 2 in Fig. 1), we propose to perform the $e.632+$ bootstrap error estimate from Efron and Tibshirani (1997) that is adequate for small sample size data sets (Ambroise and McLachlan, 2002). Note that $e.632+$ does not dictate the optimal number of features to select. The error rate estimates that are computed with respect to the number of selected variables are only a way to compare the performances of different variable selection methods. Step 2 consists in two functions called learn and evaluate. The learn function simply learns OFW on a fixed number of bootstrap samples ($B_{sample}$) with the same tuned parameters defined in step 1. The evaluate function that was inspired from the ipred package, computes and outputs the $e.632+$ error rate.
Figure 3: Probability weights with ofwCART (a) and ofwSVM (b).

Figure 4: e.632+ error rate of ofwCART and ofwSVM.
The learn and evaluate functions

```r
> learn.error.cart <- learn(srbct, as.factor(class), type="CART", ntree=150, + nforest=2500, mtry=5, Bsample=10, do.trace=100, nstable=25)
> learn.error.svm <- learn(srbct, as.factor(class), type="SVM", nsvm=30000, mtry=5, Bsample=10, do.trace=2000, nstable=25)
```

As the evaluation will be performed for a small selection size, we strongly advise to reduce
the number of total iterations, using for example the early stopping criterion.
In the literature, Bsample often equals to 10-50. On a 1.6 GHz 960 Mo RAM AMD Turion 64
X2 PC, the learning step of one bootstrap sample on a typical microarray data set \( (p \simeq 5000 \text{ and } n \simeq 50) \) can take approximatively 2.5 hours. Hence, depending on the chosen value of
Bsample, this evaluation step might be time consuming (see section 4) and one can rather
choose to perform parallel computing using the Rmpi library (see appendix).
If the SVM classifier is applied, each SVM is evaluated with the heaviest variables in \( \mathbb{P}^{b}_{nsvm} \),
which is learnt in the learn function, \( b = 1 \ldots Bsample \). If the CART classifier is applied, the
evaluate function aggregates ntreeTest trees. Each tree is constructed on a small variable
subset that is randomly selected from the heaviest variables in \( \mathbb{P}^{b}_{nforest} \), to avoid a too optimistic
evaluation (see Lé Cao et al. 2007a). Both functions evaluate the variable selection of size
maxvar:

```r
> eval.error.cart <- evaluate(learn.error.cart, ntreeTest=100, maxvar=25)
> eval.error.svm <- evaluate(learn.error.svm, maxvar=25)
```

The evalCARTparallel function has also been implemented for parallel computing (refer to
appendix).

The aim of the evaluate function is to compare the performance of several algorithms (e.g.
ofwCART and ofwSVM):

```r
> matplot(cbind(eval.error.cart$error, eval.error.svm$error), xlab="number of + selected genes", ylab="error rate e.632+", type="l", col=c(1,4), lty=c(1,4), + lwd=2, cex.lab= 1.3)
> legend(18,0.40, c("ofwCART", "ofwSVM"), col=c(1,4), lty=c(1,4), cex=1.2, + lwd=2)
```

Figure 4 displays the e.632+ bootstrap error rate of the selections resulting from either ofw-
CART or ofwSVM with respect to the number of selected genes. In this example, where we
compare the non-weighted versions of OFW, ofwCART seems to perform the best.

### 3.5 Further analysis

**Comparing the weighted and non weighted versions of OFW**

The weighting procedure presented in section 3.4 has also been included in the error evaluation
function evaluate. To compare the two approaches weighted (OFW) and non weighted
(wOFW), we strongly advise to launch the evaluate function with the argument weight=T
in both cases to evaluate if the minority classes were misclassified or not. Otherwise, the
e.632+ bootstrap error rate will always be lower for OFW than wOFW. This is illustrated in figure 5 where the same gene selection resulting from ofwCART is evaluated either with the non-weighted version of e.632+ (ofwCAR T1) or with the weighted version of e.632+ (ofwCAR T2). Even though the same two gene selections are evaluated, the error rate is lower in ofwCAR T1 as this overall error rate only takes into account the microarrays that are rightly classified in the majorities classes. In ofwCAR T2 where misclassified minoritary classes are taken into account, the error rate is consequently higher:

```r
elearn.error.cart=learn(srbct, as.factor(class), type="CART, ntree=150,
  mforest=3000, mtry=5, Bsample=10)
eval.error.cart1=evaluate(learn.error.cart, ntreeTest=100, maxvar=25)
eval.error.cart2=evaluate(learn.error.cart, ntreeTest=100, maxvar=25,
  weight=T)
matplot(cbind(eval.error.cart1$error, eval.error.cart2$error), xlab=
  "number of selected genes", ylab="error rate e.632+", type="l",
  col=c(1,1), lty=c(1,2), lwd=2, cex.lab=1.3)
legend(18,0.12, c("ofwCAR T1", "ofwCAR T2"), col=c(1,1), lty=c(1,2),
  cex=1.2, lwd=2 )
```

4 Computation time

Optimal Feature Weighting is a stochastic method that might be computationally time consuming if the variable dimension is very high. As the algorithm gets stabler for a large number of iterations, the variable selection step (step 3) might take 1-2 hours. Therefore, using parallel computing with the Rmpi library during the evaluation step (step 2) might be advisable. If the dimension is considerable, we strongly advise to pre-filter the data set so as to remove uninformative variables that slow down the computation.

In this paper, on a very small microarray data set (200 genes), the tuning step (step 1) took approximately 20 min, the evaluation step (step 2) 1.5 hour and the variable selection step (step 3) 7 min.
5 Conclusion

We have implemented the stochastic algorithm Optimal Feature Weighting to select discriminative features. Although we illustrated this method on microarray data, OFW can be applied on any continuous data set for classification and prediction purposes. Wrapper methods usually require heavy computation, and so does OFW. Efforts have thus been made to reduce some of the computation time by implementing C functions when applying CART and by proposing parallel programming during the learning step. With this package, we hope to provide the user a method with a strong theoretical background that is easy to apply and that can bring interesting results in a feature selection framework.

6 Availability and requirements

The R version ≥ 2.5.0 is needed to load the svm library e1071.

Acknowledgements

We are grateful to "Projet Calcul en MIdi-Pyrenées" (CALMIP) for the intensive computations.

References


Appendix

Parallel computing with ofwCART

## This is an example to perform the learning and the evaluation step of ofwCART with parallel computing
## A part of this code has been borrowed from Rmpi examples of the Acadia Center for Mathematical Modelling and Computation
## http://ace.acadiau.ca/math/ACMMaC/Rmpi/examples.html

library("Rmpi")
library(rlecuyer)
library(e1071, lib.loc="MyR/Library") #if the library e1071 is locally installed
library(ofw, lib.loc="MyR/Library") #if the library ofw is locally installed

mpi.spawn.Rslaves(nslaves=5) #number of slaves to spawn, should be equal to B, where B = the number of bootstrap samples
mpi.setup.rngstream() #generates random numbers

.Last <- function(){
  if (is.loaded("mpi_initialize")){
    if (mpi.comm.size(1) > 0){
      print("Please use mpi.close.Rslaves() to close slaves.")
      mpi.close.Rslaves()
    }
    print("Please use mpi.quit() to quit R")
    .Call("mpi_finalize")
  }
}

##----------FUNCTION ------------------------------------------
## learn ofw on the bootstrap sample
## assume all the parameters
learn.ofw = function(){
  #if both libraries are locally installed:
  library(e1071, lib.loc="MyR/Library")
  library(ofw, lib.loc="MyR/Library")

  x=data[mat.train[,foldNumber],]
  y=class[mat.train[,foldNumber]]
  res=ofw(x=x, y=as.factor(y), type=type, ntree=ntree, nforest=nforest, nsvm=nsvm, mtry=mtry, do.trace=do.trace, nstable=nstable, weight=weight)
  return(list(res$prob))
}
## MAIN

```r
# We are in the parent.
# read the data set here attached in the library of w

data(srbct)
attach(srbct)
data=srbct
class=class

# define parameters and constants
B=5 # number of bootstrap samples
nvar=ncol(data)
nobs=nrow(data)
mtry=5
do.trace=FALSE
nstable=FALSE
weight=F

# parameters to learn and evaluate of w CART:
type= "CART"
ntree=10
nforest=10

# during evaluation
maxvar = 10
ntreeTest = 15

# define the matrices
mat.train = matrix(nrow=nobs, ncol=B)
mat.P= matrix(nrow=nvar, ncol=B)

# define the bootstrap samples
for(i in 1:B)
{
  again=T
  while(again){
    train=sample(1:nobs, nobs, replace=T)
    if(any(table(class[train])==0)) {again=T} else {again=F}
  }
  mat.train[,i]=train
}

nslaves= mpi.comm.size() -1
```
#send parameters and constants to slaves
mpi.bcast.Robj2slave(data)
mpi.bcast.Robj2slave(class)
mpi.bcast.Robj2slave(B)
mpi.bcast.Robj2slave(nvar)
mpi.bcast.Robj2slave(nobs)
mpi.bcast.Robj2slave(mtry)
mpi.bcast.Robj2slave(do.trace)
mpi.bcast.Robj2slave(nstable)
mpi.bcast.Robj2slave(weight)
mpi.bcast.Robj2slave(type)
mpi.bcast.Robj2slave(ntree)
mpi.bcast.Robj2slave(maxvar)
mpi.bcast.Robj2slave(ntreeTest)
mpi.bcast.Robj2slave(nforest)
mpi.bcast.Robj2slave(mat.train)
mpi.bcast.Robj2slave(mat.P)

#send functions to slaves
mpi.bcast.cmd(foldNumber <- mpi.comm.rank())
mpi.bcast.Robj2slave(learn.ofw)

#each slave learns ofwCART on each bootstrap sample
res.slaves = mpi.remote.exec(learn.ofw(), comm=1)

#get the results
for (i in 1:nslaves)
{
  mat.P[,i]=res.slaves[[i]][[1]]
}

#once the probability has been learnt on each bootstrap sample,
#evaluate the selection with the function evaluateCARTparallel.R
res.eval=evaluateCARTparallel(x=data, y=as.factor(class), matTrain = mat.train,
matProb= mat.P, maxvar = maxvar, ntreeTest=ntreeTest, weight=weight)

#write output
write.table(mat.train, "mat.train.txt")
write.table(mat.P, "mat.P.txt")
write.table(res.eval$error, "error.txt")
detach(srbct)

mpi.close.Rslaves()
mpi.quit(save="yes")