GARS: a Genetic Algorithm for the identification of Robust Subsets of variables in high-dimensional and challenging datasets

***Mattia Chiesa*[1], *Giada Maioli*[2], *and Luca Piacentini*[1]**

[1]Immunology and Functional Genomics Unit, Centro Cardiologico Monzino, IRCCS, Milan, Italy;
[2]Universitá degli Studi di Pavia, Pavia, Italy

May 3, 2019

Abstract

Feature selection aims to identify and, remove redundant, irrelevant and noisy variables from high-dimensional datasets. Selecting informative features affects the subsequent classification and regression analyses by improving their overall performances. Several methods have been proposed to perform feature selection: most of them relies on univariate statistics, correlation, entropy measurements or the usage of backward/forward regressions.
Herein, we propose an efficient, robust and fast method that adopts stochastic optimization approaches for high-dimensional. Genetic algorithms, a type of evolutionary algorithms, are often used to find solutions for optimization and search problems and promise to be effective on complex data. They operate on a population of potential solutions and apply the "principle of survival of the fittest" to produce the better approximation of the optimal solution.
GARS is an innovative implementation of a genetic algorithm that selects robust features in high-dimensional and challenging datasets.

Package

GARS 1.4.0

Contents

1 Introduction

A crucial step in a Data Mining analysis is Feature Selection, which is the process of identifying the most informative predictors to build accurate classification models. Indeed, many features could inadvertently introduce bias in a prediction model, lead to overfitting and increase the complexity in further downstream analysis. In last decades, several methods have been proposed to perform feature selection; they are usually grouped in three main classes [1, 2]:

- **Filter Methods** - A measure based on statistics or entropy is used to rank an then select variables. The most popular filter methods are the *Information Gain*, the *ReliefF*, the *Gini Index* and the χ^2 test;
- **Wrapper Methods** - The set of important features is identified using a classifier to evaluate the accuracy of several combinations of variables. *Backward/Forward/Stepwise-elimination* stategies belong to this category;
- **Embedded Methods** - As for wrapper methods, a classifier is used to identify the best set of variables; however, in this case the procedure to identify the features is totally joined with the construction of the classifier. The "tree-based" classifiers (e.g. *Decision trees* and the *Random Forest*) are the widely used embedded methods.

Genetic Algorithms (GAs) are heuristic adaptive search algorithms that simulate the Darwinian law of "the survival of the fittest" among individuals, over consecutive generations, for solving hard problems, such as pattern recognition and feature selection. A GA is traditionally composed of three main consecutive stages: first, a random set of candidate solutions, *i.e.* chromosomes, is generated. Then, each chromosome is evaluated by a custom score, *i.e. fitness function* that reflects how good a solution is. Finally, the evolutionary operators are sequentially applied to the entire population: *Selection*, *Crossover* and *Mutation*. To find the optimal solution, this process has to be repeated several times: the starting chromosome population of a certain generation corresponds to the resulting chromosome population of the previous generation.
The idea to use a GA to perform Feature Selection is not novel; however, all the developed GA-based methods needs a classifier to evaluate the goodness of a set of features (namely, they belong to the "Wrapper" or the "Embedded" category). The classifiers mainly used to assess the selected features are the Support Vector Machines (GA-SVN) [3], the k-Nearest Neighbours (GA-KNN) [4], the Random Forest (rfGA, see the *caret* package for details), the LDA (caretGA, see the *caret* package for details) [5] and maximum-likelihood based methods (GA-MLHD) [6].
One of the most relevant contexts where the feature selection is becoming more and more essential, is the *-OMICs* field: in fact, datasets coming from genomics, transcriptomics, proteomics and metabolomics experiments are typically composed of a large number of features compared to the sample size; this poses a big challenge for a data mining analysis.
In this context, we developed an innovative implementation of a **G**enetic **A**lgorithm that selects **R**obust **S**ubsets of features (**GARS**) in high-dimensional and challenging datasets. GARS has several benefits:

1. it does not need any classifier to evaluate the goodness of the selected feature. The fitness is calculated by the averaged Silhouette Index [7], after computing a Multi-Dimensional Scaling of the data. This allows being less prone to overfitting and local optima;
2. it is relatively fast, when the number of features is relatively high (tens of thousands);
3. it can be used in multi-class classifcation problems;

4. even though it has been thought for solving -*OMICs* tasks, it can be easily used in several other contexts (e.g. low-dimensional data);
5. it can be easily integrated with other R and Bioconductor packages for Data Mining (e.g. *caret* and *DaMiRseq*).

2 Using GARS: a classification analysis

2.1 The testing RNA-Seq dataset

The dataset used in this vignette comes from a miRNA-Seq experiment performed on cervical tissues [8]; the dataset is composed of 714 miRNAs and 58 samples: 29 Tumor (T) and 29 Non-Tumor (N) cervical samples, respectively. In order to obtain a normalized gene expression matrix, we used the `DaMiR.normalization` function of the *DaMiRseq* package with default parameters.

```
library(MLSeq)
library(DaMiRseq)
library(GARS)

# load dataset
filepath <- system.file("extdata/cervical.txt", package = "MLSeq")
cervical <- read.table(filepath, header=TRUE)

# replace "wild-card" characters with other characters
rownames(cervical) <- gsub("*", "x", rownames(cervical), fixed = TRUE)
rownames(cervical) <- gsub("-", "_", rownames(cervical), fixed = TRUE)

# create the "class" vector
class_vector <- data.frame(gsub('[0-9]+', '', colnames(cervical)))
colnames(class_vector) <- "class"
rownames(class_vector) <- colnames(cervical)

# create a Summarized Experiment object
SE_obj <- DaMiR.makeSE(cervical, class_vector)

## Your dataset has:
## 714 Features;
## 58 Samples, divided in:  29 N 29 T
## 1 variables: class ;
##  'class' included.

# filter and normalize the dataset
datanorm <- DaMiR.normalization(SE_obj)

## 545 Features have been filtered out by espression. 169 Features remained.
## 8 'Hypervariant' Features have been filtered out. 161 Features remained.
## Performing Normalization by 'vst' with dispersion parameter:  parametric
```

2.2 Launch GARS

After filtering and normalizing data we got a dataset with 161 expressed miRNAs and 58 samples. The best way to use *GARS* for selecting a robust set of features from an high-dimentional dataset is to exploit the wrapper function `GARS_GA`. A dataset must be provided, as well as a vector containing the class information. *GARS* gives the opportunity to provide

input data in the form of `SummarizedExperiment`, `matrix` or `data.frame` objects. On one hand, `SummarizedExperiment` is preferred in the case of a RNA-Seq experiment; on the other hand, a `data.frame` allows the user to integrate expression data with other numerical and/or categorical features.

In addition, several other parameters have to be set:

- `chr.num` - The number of chromosomes in each population. If the number of chromosomes is too small, the GA will explore a small part of the "solution space" (each chromosome is a candidate solution); conversely, if the number is too high, the GA will produce results very slowly. Default: 1000
- `chr.len` - The length of each chromosome. **This argument is the most important in *GARS*: it corresponds to the length of the desired feature set**. Usually, in data mining analysis the number of features, needed to build a classification model, has to be much smaller than the number of observations (i.e. samples).
- `generat` - The maximum number of generations. This number is usually high (hundreds to thousands): the higher the number of generations, the higher the probability to reach the best solution. Default: 500
- `co.rate` - The probability to perform the crossover for each random couple of chromosomes. This parameter allows the evolution rate to be controlled and tuned. Default: 0.8
- `mut.rate` - The probability to mutate each chromosome base. This parameter allows the evolution rate to be controlled and tuned. Default: 0.01
- `n.elit` - The number of best chromosomes that must be "preserved from the evolution". This number is usually small compared to the number of chromosomes, in order to enhance the evolution. Default: 10
- `type.sel` - The algorithm that performs the Selection step. *"Roulette Wheel"* and *"Tournament"* selections are implemented. Default: *Roulette Wheel*.
- `type.co` - The algorithm that performs the Crossover step. *"One point"* and *"Two points"* crossover are implemented. Default: *One point*.
- `type.one.p.co` - In the case of *"One point"* crossover, this argument allows setting the quartile where the crossover has to be applied. The user can choose among the first, the second and the third quartile. Default: First quartile.
- `n.gen.conv` - The maximum number of consecutive generations with the same maximum fitness score. When the maximum fitness scores are the same for several generation, this means that the GA found the optimal solution (i.e. reached the convergence). This argument is useful to stop *GARS* when the convergence is reached. Default: 80.
- `plots` - Whether generating plots or not. Default: yes.
- `verbose` - Whether printing information in the console or not. Default: yes.
- `n.Feat_plot` - If `plots = yes`, the number of features to be plotted by `GARS_PlotFeaturesUsage`

To speed up the execution time of the function, here we set `generat = 20`, `chr.num = 100` and `chr.len = 8`; however, for a typical -omic experiment (thousands of features), this is probably not sufficent to find the best feature set. We strongly recomend to set accurately each parameter, trying different combinations of them (especially `chr.len`, See Section 2.4).

```
set.seed(123)
res_GA <- GARS_GA(data=datanorm,
                  classes = colData(datanorm),
                  chr.num = 100,
                  chr.len = 8,
                  generat = 20,
                  co.rate = 0.8,
                  mut.rate = 0.1,
                  n.elit = 10,
                  type.sel = "RW",
                  type.co = "one.p",
                  type.one.p.co = "II.quart",
                  n.gen.conv = 150,
                  plots="no",
                  verbose="yes")
```

```
## GARS has been set with these parameters:
##  Number of starting features: 161
##  Number of samples: 58
##  Number of classes: 2
##  Number of chromosomes: 100
##  Length of chromosomes (i.e. number of desired features): 8
##  Number of maximum generations: 20
##  Crossing-Over rate: 0.8
##  Mutation rate: 0.1
##  Number of chromosomes kept by elitism: 10
##  Type of Selection method: RW
##  Type of CrossingOver method: one.p
##  Position of the one-Point Crossover: II.quart
##  Number of max generations allowed with the same Fitness: 150
##  Produce graphs: no
##
##  ##############################
##  ####    GARS is running   ####
##  ##############################
##
## Reached 10 iterations...
## Reached 20 iterations...
## GARS found a solution after 20 iterations.
## With these parameters, the best solution:
##
##  1. is reached after 20 iterations;
##  2. is reached looking at the 100 % of the 161 features;
##  3. got a maximum fitness score = 0.55
##  4. is composed of the following features:
##  miR_21 miR_146a let_7i miR_339_5p miR_7 miR_125a_5p miR_204 miR_142_5p
```

The results of `GARS_GA` are stored in a `GarsSelectedFeatures` object, herein `res_GA`, where the informations could be extracted by 4 Assessor methods:

- `MatrixFeatures()` - Extracts the `matrix` containing the expression values for the selected features;

- `LastPop()` - Extracts the `matrix` containing the chromosome population of the last generation. The first column of this matrix represent the best solution, found by the GA;
- `AllPop()` - Extracts the `list` containing all the populations produced over the generations;
- `FitScore()` - Extracts the `vector` containing the maximum fitness scores, computed in each generation.

The information stored in the `GarsSelectedFeatures` object could be used for downstream analysis (See Section 2.3) or for generating plots (before, we set `plots = no`). In *GARS* the functions `GARS_PlotFitnessEvolution()` allows the user to plot the fitness evolution over the generations, while the function `GARS_PlotFeaturesUsage()` allows representing the frequency of each feature in a bubble chart:

```
# Plot Fitness Evolution
fitness_scores <- FitScore(res_GA)
GARS_PlotFitnessEvolution(fitness_scores)
#Plot the frequency of each features over the generations
Allfeat_names <- rownames(datanorm)
Allpopulations <- AllPop(res_GA)
GARS_PlotFeaturesUsage(Allpopulations,
                       Allfeat_names,
                       nFeat = 10)
```

As mentioned before, in this example the number of chromosomes (100) and the number of generations (20) were intentionally small. Nevertheless, the population evolved over the generations: indeed, as shown in Figure 1, the maximum fitness score is equal to 0.41 in the first generation and reaches the value of 0.55 in the last generation (an increasing of 30%). Moreover, the Figure 2 shows the most recurring (i.e. "conserved") miRNAs over the iterations.

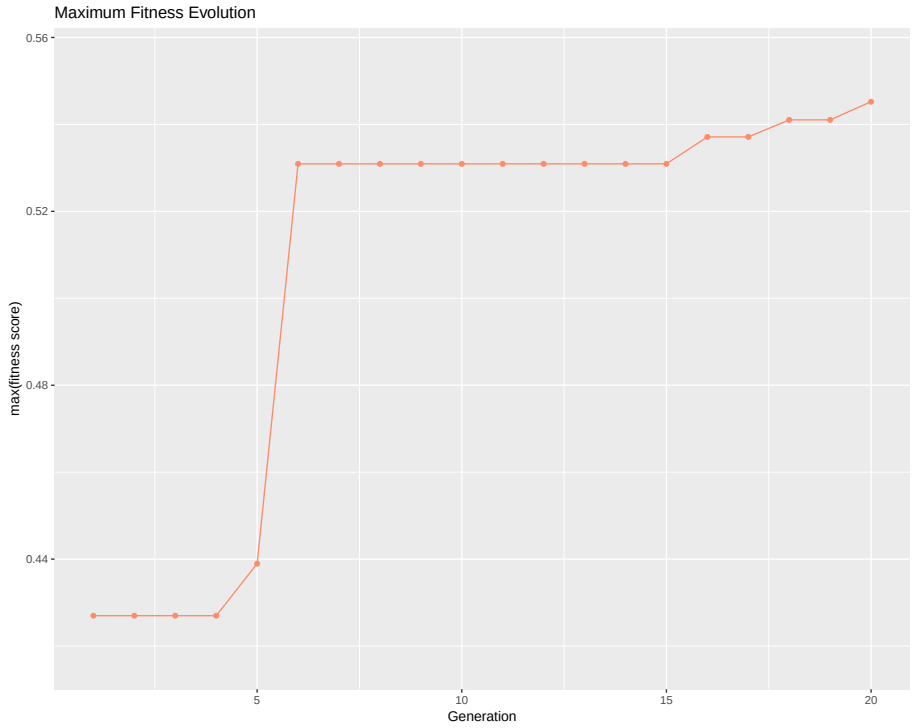

Figure 1: **Fitness Evolution plot**
The plot shows the evolution of the maximum fitness across the generations

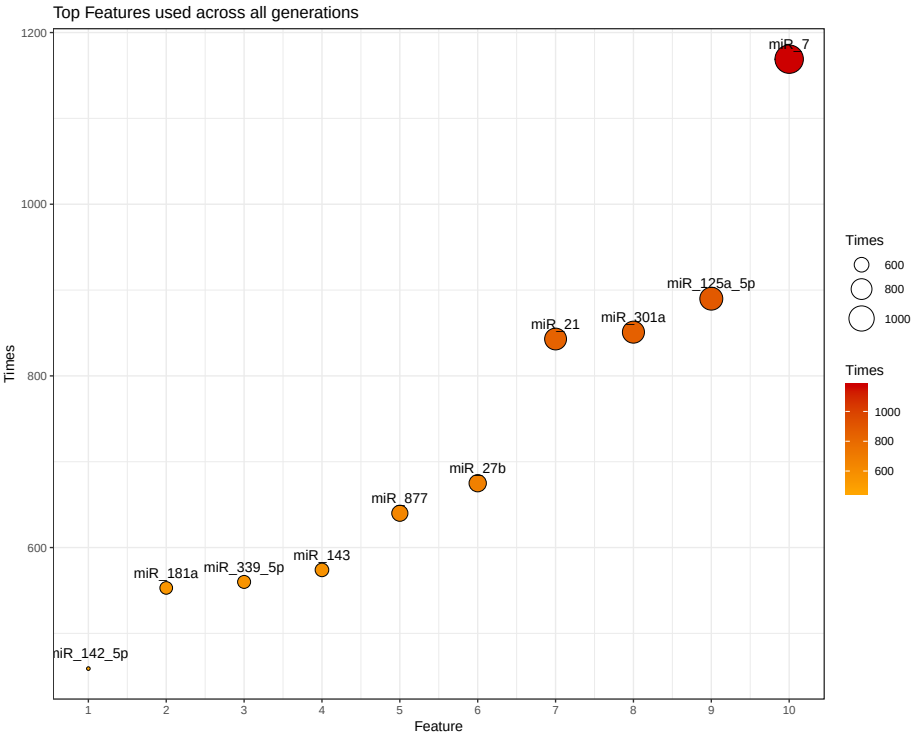

Figure 2: **Recurring Features**
Each circle in the plot represents a feature. The color and size of each circle are, respectively, darker and bigger when a feature is more recurring.

2.3 Test the robustness of the feature set

Besides the maximum fitness score of the last population, we can assess the quality of the results, through a classification analysis. To perform this task, we used the functions implemented in the *DaMiRseq* package, which offers several easy-to-use and efficient functions for Data Mining; however, the user may perform the analysis, exploiting other packages for data mining, such as the *caret* package.
First, we extracted data from the `MatrixFeatures(res_GA)` object where the expression values for the selected features are stored. Then,we tranformed this matrix and the `classes_GARS` in a `data.frame` object that we used as input for the `DaMiR.EnsembleLearning` function. We set `iter = 5` for practical reasons.

```
# expression data of selected features
data_reduced_GARS <- MatrixFeatures(res_GA)

# Classification
data_reduced_DaMiR <- as.data.frame(data_reduced_GARS)
classes_DaMiR <- as.data.frame(colData(datanorm))
colnames(classes_DaMiR) <- "class"
rownames(classes_DaMiR) <- rownames(data_reduced_DaMiR)

DaMiR.MDSplot(data_reduced_DaMiR,classes_DaMiR)
DaMiR.Clustplot(data_reduced_DaMiR,classes_DaMiR)
set.seed(12345)
Classification.res <- DaMiR.EnsembleLearning(data_reduced_DaMiR,
                                             classes_DaMiR$class,
                                             iter=5)

## You select: RF LR kNN LDA NB SVM weak classifiers for creating
##       the Ensemble meta-learner.
## Ensemble classification is running.  5  iterations were chosen:
## Accuracy [%]:
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.94 0.94 0.93 0.92 0.93 0.92 0.91
##  St.Dev. 0.04 0.04 0.05 0.05 0.02 0.03 0.06
## MCC score:
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.89 0.89 0.87 0.85 0.87 0.85 0.84
##  St.Dev. 0.08 0.08 0.09 0.1 0.04 0.06 0.11
## Specificity:
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.94 0.94 0.96 0.9 0.94 0.92 0.96
##  St.Dev. 0.06 0.06 0.06 0.06 0.05 0.08 0.06
## Sensitivity:
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.96 0.96 0.92 0.96 0.94 0.94 0.89
##  St.Dev. 0.06 0.06 0.08 0.06 0.08 0.06 0.11
## PPV:
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.93 0.93 0.96 0.89 0.93 0.91 0.96
##  St.Dev. 0.06 0.06 0.06 0.08 0.06 0.09 0.06
## NPV:
```

```
##  Ensemble RF SVM NB LDA LR kNN
##  Mean: 0.96 0.96 0.91 0.96 0.93 0.93 0.87
##  St.Dev. 0.06 0.06 0.09 0.06 0.1 0.06 0.14
```

The features selected by *GARS* allowed us to clearly discriminate the N and T classes (See Figures 3 and 4). Moreover, we obtained high classification accuracy for all the classifiers built using the features set (See Figure 5).

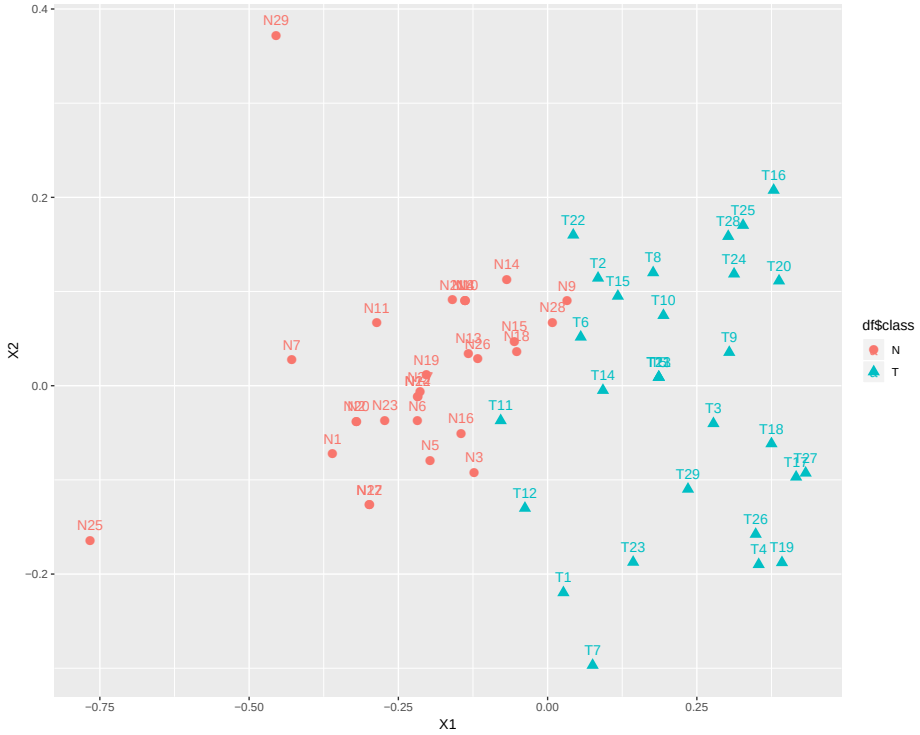

Figure 3: MultiDimensional Scaling plot
The MDS is drawn using the 8 features selected by *GARS*. The averaged Silhouette Index (i.e. the maximum fitness function of the last population) is equal to 0.55.

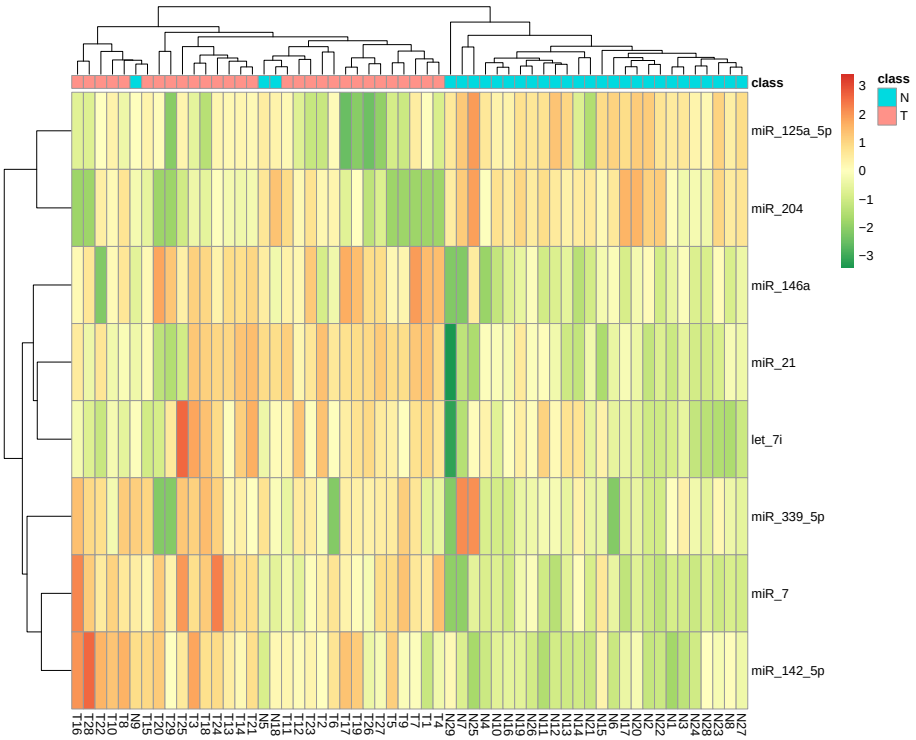

Figure 4: Clustergram
The clustergram is drawn using the 8 features selected by *GARS*.

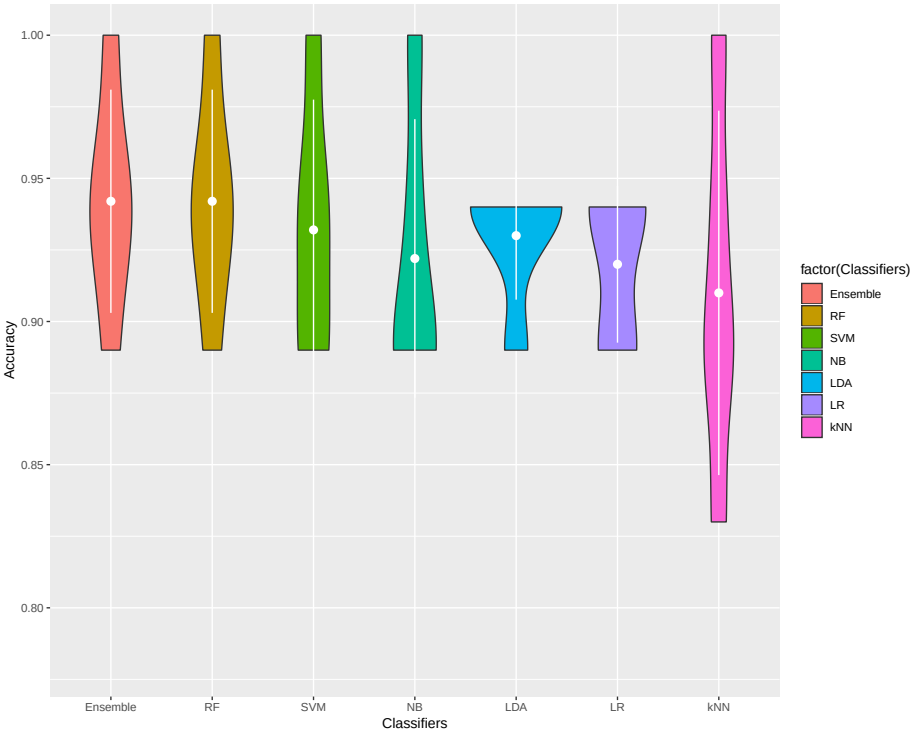

Figure 5: Violin plot
The violin plot highlights the classification accuracy of each classifier. Using the features set, selected by *GARS*, the averaged classification accuracy is always high, dispite the small number of iterations.

2.4 Find the best features set

In the previous Section, we run *GARS* setting `chr.len = 8`. In this way, we forced the algorithm to find the best solution consisting of 8 features. However, this is probably not the best solution ever, but rather the optimal solution with 8 features. In order to find the best solution, we need to try several values of `chr.len`.
A practical solution is to insert the `GARS_GA` function inside a for loop. In the next example, we run *GARS* with `chr.len` equal to 7, 8 and 9. Finally, we search for the best solution.

```
populs <- list()
k=1
for (ik in c(7,8,9)){
  set.seed(1)
  cat(ik, "features","\n")
  populs[[k]] <- GARS_GA(data=datanorm,
                         classes = colData(datanorm),
                         chr.num = 100,
                         chr.len = ik,
                         generat = 20,
                         co.rate = 0.8,
                         mut.rate = 0.1,
                         n.elit = 10,
                         type.sel = "RW",
                         type.co = "one.p",
                         type.one.p.co = "II.quart",
                         n.gen.conv = 150,
                         plots = "no",
                         verbose="no")

  k <- k +1

}
```

```
## 7 features
## Reached 10 iterations...
## Reached 20 iterations...
## GARS found a solution after 20 iterations.
## 8 features
## Reached 10 iterations...
## Reached 20 iterations...
## GARS found a solution after 20 iterations.
## 9 features
## Reached 10 iterations...
## Reached 20 iterations...
## GARS found a solution after 20 iterations.
```

```
# find the maximum fitness for each case
max_fit <- 0

for (i in seq_len(length(populs))){
  max_fit[i] <- max(FitScore(populs[[i]]))
}
```

```
max_fit
```

```
## [1] 0.5438938 0.5417438 0.5519847
```

```
best_popul <- populs[[which(max_fit == max(max_fit))]]

# number of features (best solution)
dim(MatrixFeatures(best_popul))[2]
```

```
## [1] 9
```

Now, we can compare the results obtained from several `chr.len` values and select the best solution, looking at the maximum fitness scores and, eventually, applying the "law of parsimony" principle (*Occam's razor*).

3 Build your custom GA

As mentioned in Section 2, the best way to use *GARS* is to run the `GARS_GA` function. However, the *GARS* package allows the user to build a custom GA (e.g. avoiding the Crossover step), joining the functions embedded in `GARS_GA`:

- `GARS_create_rnd_population()` - allows creating a random chromosome population;
- `GARS_FitFun()` - allows computing the fitness function, given a chromosome population;
- `GARS_Elitism()` - allows splitting a chromosome population, ordered by fitness scores;
- `GARS_Selection()` - allows selecting the best chromosomes, given a chromosome population;
- `GARS_Crossover()` - allows performing the Crossover step;
- `GARS_Mutation()` - allows performing the Mutation step;
- `GARS_PlotFitnessEvolution()` - allows plotting the evolution of the maximum fitness over the generations;
- `GARS_PlotFeaturesUsage()` - allows plotting how many times a feature is present over the generations;

4 Session Info

- R version 3.6.0 (2019-04-26), `x86_64-w64-mingw32`
- Locale: `LC_COLLATE=C`, `LC_CTYPE=English_United States.1252`, `LC_MONETARY=English_United States.1252`, `LC_NUMERIC=C`, `LC_TIME=English_United States.1252`
- Running under: `Windows Server 2012 R2 x64 (build 9600)`
- Matrix products: default
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: Biobase 2.44.0, BiocGenerics 0.30.0, BiocParallel 1.18.0, DaMiRseq 1.8.0, DelayedArray 0.10.0, GARS 1.4.0, GenomeInfoDb 1.20.0, GenomicRanges 1.36.0, IRanges 2.18.0, MLSeq 2.2.0, S4Vectors 0.22.0, SummarizedExperiment 1.14.0, caret 6.0-84, cluster 2.0.9, ggplot2 3.1.1, knitr 1.22, lattice 0.20-38, matrixStats 0.54.0
- Loaded via a namespace (and not attached): AnnotationDbi 1.46.0, BiocManager 1.30.4, BiocStyle 2.12.0, Biostrings 2.52.0, DBI 1.0.0, DESeq 1.36.0, DESeq2 1.24.0, EDASeq 2.18.0, FSelector 0.31, FactoMineR 1.41, Formula 1.2-3, GenomeInfoDbData 1.2.1, GenomicAlignments 1.20.0, GenomicFeatures 1.36.0, Hmisc 4.2-0, MASS 7.3-51.4, Matrix 1.2-17, ModelMetrics 1.2.2, R.methodsS3 1.7.1, R.oo 1.22.0, R.utils 2.8.0, R6 2.4.0, RColorBrewer 1.1-2, RCurl 1.95-4.12, RSNNS 0.4-11, RSQLite 2.1.1, RWeka 0.4-40, RWekajars 3.9.3-1, Rcpp 1.0.1, Rsamtools 2.0.0, ShortRead 1.42.0, XML 3.98-1.19, XVector 0.24.0, abind 1.4-5, acepack 1.4.1, annotate 1.62.0, arm 1.10-1, aroma.light 3.14.0, assertthat 0.2.1, backports 1.1.4, base64enc 0.1-3, bdsmatrix 1.3-3, biomaRt 2.40.0, bit 1.1-14,

bit64 0.9-7, bitops 1.0-6, blob 1.1.1, boot 1.3-22, checkmate 1.9.1, class 7.3-15, coda 0.19-2, codetools 0.2-16, colorspace 1.4-1, compiler 3.6.0, corrplot 0.84, crayon 1.3.4, data.table 1.12.2, digest 0.6.18, dplyr 0.8.0.1, e1071 1.7-1, edgeR 3.26.0, entropy 1.2.1, evaluate 0.13, flashClust 1.01-2, foreach 1.4.4, foreign 0.8-71, genalg 0.2.0, genefilter 1.66.0, geneplotter 1.62.0, generics 0.0.2, glue 1.3.1, gower 0.2.0, grid 3.6.0, gridExtra 2.3, gtable 0.3.0, highr 0.8, hms 0.4.2, htmlTable 1.13.1, htmltools 0.3.6, htmlwidgets 1.3, httr 1.4.0, hwriter 1.3.2, igraph 1.2.4.1, ineq 0.2-13, ipred 0.9-9, iterators 1.0.10, kknn 1.3.1, labeling 0.3, latticeExtra 0.6-28, lava 1.6.5, lazyeval 0.2.2, leaps 3.0, limma 3.40.0, lme4 1.1-21, locfit 1.5-9.1, lubridate 1.7.4, magrittr 1.5, memoise 1.1.0, mgcv 1.8-28, minqa 1.2.4, munsell 0.5.0, mvtnorm 1.0-10, nlme 3.1-139, nloptr 1.2.1, nnet 7.3-12, pheatmap 1.0.12, pillar 1.3.1, pkgconfig 2.0.2, pls 2.7-1, plsVarSel 0.9.4, plyr 1.8.4, prettyunits 1.0.2, prodlim 2018.04.18, progress 1.2.0, purrr 0.3.2, rJava 0.9-11, randomForest 4.6-14, recipes 0.1.5, reshape2 1.4.3, rlang 0.3.4, rmarkdown 1.12, rpart 4.1-15, rstudioapi 0.10, rtracklayer 1.44.0, sSeq 1.22.0, scales 1.0.0, scatterplot3d 0.3-41, splines 3.6.0, stringi 1.4.3, stringr 1.4.0, survival 2.44-1.1, sva 3.32.0, tibble 2.1.1, tidyselect 0.2.5, timeDate 3043.102, tools 3.6.0, withr 2.1.2, xfun 0.6, xtable 1.8-4, yaml 2.2.0, zlibbioc 1.30.0

References

[1] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[2] Zena M Hira and Duncan F Gillies. A review of feature selection and feature extraction methods applied on microarray data. *Advances in bioinformatics*, 2015, 2015.

[3] Mohd Saberi Mohamad, Safaai Deris, and Rosli Md Illias. A hybrid of genetic algorithm and support vector machine for features selection and classification of gene expression microarray. *International Journal of Computational Intelligence and Applications*, 5(01):91–107, 2005.

[4] Leping Li, Clarice R Weinberg, Thomas A Darden, and Lee G Pedersen. Gene selection for sample classification based on gene expression data: study of sensitivity to choice of parameters of the ga/knn method. *Bioinformatics*, 17(12):1131–1142, 2001.

[5] Max Kuhn et al. The caret package. *Journal of Statistical Software*, 28(5):1–26, 2008.

[6] CH Ooi and Patrick Tan. Genetic algorithms applied to multi-class prediction for the analysis of gene expression data. *Bioinformatics*, 19(1):37–44, 2003.

[7] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[8] Daniela Witten, Robert Tibshirani, Sam G Gu, Andrew Fire, and Weng-Onn Lui. Ultra-high throughput sequencing-based small rna discovery and discrete statistical biomarker analysis in a collection of cervical tumours and matched controls. *BMC biology*, 8(1):58, 2010.