dGAselID: An R Package for Selecting a Variable Number of Features in High Dimensional Data

by Nicolae Teodor Melita and Stefan Holban

Abstract The **dGAselID** package proposes an original approach to feature selection in high dimensional data. The method is built upon a diploid genetic algorithm. The genotype to phenotype mapping is modeled after the Incomplete Dominance Inheritance, overpassing the necessity to define a dominance scheme. The fitness evaluation is done by user selectable supervised classifiers, from a broad range of options. Cross validation options are also accessible. A new approach to crossover, inspired from the random assortment of chromosomes during meiosis is included. Several mutation operators, inspired from genetics, are also proposed. The package is fully compatible with the data formats used in **Bioconductor** and **MLInterfaces** package, readily applicable to microarray studies, but is flexible to other feature selection applications from high dimensional data. Several options for the visualization of evolution and outcomes are implemented to facilitate the interpretation of results. The package's functionality is illustrated by examples.

Introduction

Recent advances in information technology provide tools for gathering an immense amount of data on various scopes. Investigators have increasingly improved tools to collect data describing different areas of research. The need to efficiently manage, analyze and extract the important features from high dimensional data is also growing with the different exploration areas benefiting from these technologies.

The DNA microarray technology is widely used for exploring the differential gene expression. The method is very established and offers a very good opportunity to develop new methods for selecting features in real high dimensional data. The vast amount of microarray data that is freely available along with the results obtained by employing other exploratory techniques, belonging to statistics or artificial intelligence, provides a unique opportunity to evaluate the performance of newly developed techniques.

The Genetic Algorithms (GAs) were extensively used to select features in various high dimensional data, for different research goals. The GA designs evolved and were adapted with particular exploration interests since they were introduced (Holland, 1975). Different GA designs were specifically adapted to address optimizations or diverse feature selection (Xue et al., 2016) assignments.

The literature on GAs is comprehensive and covers various aspects of interest. The fundamentals of GAs, including the schema theorem, are covered in very instructive introductory books (Mitchell, 1998) and (Goldberg, 1989). Other authors propose exhaustive investigations in the GAs' behavior (Berard and Bienvenue, 2003) and properties (Rudolph, 1994). The genetic operators and their impact on evolution, with emphasis on mathematical details (Doerr and Doerr, 2015) were extensively examined. The genetic algorithms model the naturally occurring evolution and are designed to solve particular problems. In consequence, the theoretical foundations are yet to catch up with the practically applied algorithms. Nevertheless, the theory of genetic algorithms is emerging (Droste et al., 2002).

The project R offers a great environment for developing methods for high dimensional data analysis. The variety of techniques already implemented by numerous contributors and the availability of the methods, source code and countless data and results, as well as the very forthcoming community make it the environment of choice for implementing our method. Moreover, the Bioconductor project (Huber et al., 2015) available in R, offers a wide range of methods and tools for analyzing microarray data, as well as real data sets to experiment with and compare the results. Our package **dGAselID** was developed to be cohesive with Bioconductor. The `"ExpressionSet class"` used in Bioconductor was adopted as standard for our package; any data formatted accordingly can be analyzed with our method.

Different GA implementations are available as contributors' packages in R. Implementations of GAs for both floating-point and binary chromosomes are included in the **genalg** (Willighagen and Ballings, 2015) package. The **GA** (Scrucca, 2016), **nsga2R** (Tsou, 2015), and **gaoptim** (Tenorio, 2013) packages are dedicated to optimizations using GAs. A GA designed for determining training populations (Akdemir et al., 2015) is offered in the **STPGA** package. The package **kofnGA** (Wolters, 2015) aims to select a fixed-size set of integers. Variable selection applications of GAs are proposed in the

mogavs (Pajala, 2016) and **gaselect** (Kepplinger, 2015) packages for regression and high-dimensional data respectively.

Algorithm

Haploid GAs were previously employed to address feature selection in microarray studies (Melita et al., 2008). In this type of data, the number of samples is significantly lower than the number of features and the utilization of cross validation techniques is necessary for reliable results. The diploid GAs offer better performance than the haploid implementations for selecting features in a cross validation scenario in general, and for microarray data (Melita and Holban, 2016b) in particular.

The GA implementation in the **dGAselID** package uses a diploid representation. All the features in the data form a genome, and each feature retains a specific locus in the genome, the position in the original data. Every individual in the population will consist of two such genomes.

The fitness evaluation function is a supervised classifier. Any implementation of supervised classifier available in **MLInterfaces** package (Carey et al., 2016) is a possible choice for fitness evaluation function in **dGAselID**. The fitness value is the accuracy of the given classifier in discerning between samples belonging to different classes.

Every feature in the data, inputted according to the format in the `"ExpressionSet class"`, is represented by a gene in the genome. Every gene has two alleles, represented as `0` and `1`. The allele `1` codes for the corresponding feature to be present in the classifier. The allele `0` cyphers for discarding the gene from the classifier. A genome with a limited number of alleles = `1`, codes for the supervised classifier working on a subset of features from the data. The number of desired features is user selectable at the initialization of the algorithm.

Our implementation offers the possibility to divide the genomes into a variable number of chromosomes. The number of chromosomes to split the genomes in, is user selectable. The value `1` for the number of chromosomes will result in the genome being treated as a single chromosome, like in the classical GA implementation. The default value in the **dGAselID** is `22`. In this case, the genome is parted into 22 chromosomes, the total of human autosomes. The chromosomes will have variable length, with different number of genes, following the dispersal found in the human autosomes, as illustrated in the Table 1. The number of genes found on each chromosome will follow the spread found in the human autosomes with different values for the number of chromosomes. We chose the default value `22` to emphasize the foundation of our evolutionary approach. Different values will serve diverse practical applications. This parameter is particularly important when variables belong to several previously known categories, as with the custom microarray chips. When no such information is known, an appropriate value can be empirically determined.

The initial population is randomly generated from a discrete uniform distribution. The user can specify the number of genomes in the population, the number of activated genes in each genome and the number of chromosomes to split the genomes in. The population will encompass individuals, with each individual consisting of two sets of haploid chromosomes, randomly assigned.

In a diploid GA it is mandatory to determine how different alleles on heterozygous chromosomes influence the phenotype. The dominance schemes typically used in genetic algorithms are built upon the Complete Dominance model, described in biology by Gregor Mendel in 1865. In this model, one of the alleles, called dominant, produces effects into phenotype and masks the existence of the other allele in genotype. The alternative that does not affect the phenotype is called recessive allele. The Complete Dominance model describes only a few of the interactions between alleles in nature. Various models were later developed to describe different interactions between alleles. In Incomplete Dominance model, the phenotype of an individual is considered to be in between the phenotypes resulting from each of the inherited alleles. Both alleles influence the phenotype and the existence of none is masked in genotype. The main difference between the two models is illustrated in Figure 1. We can suppose that a gene that codes for the color of an organism has two alleles; one of them produces a red individual and the alternative shapes a blue entity. With the Complete Dominance model, one of the alleles is dominant and masks the presence of the other. In our example, the allele that codes for the red color is dominant and is noted with capital letter (R). Every diploid organism that inherits at least one allele R will be a red entity. Only if a diploid organism inherits two copies of the recessive allele (b), the phenotype will be influenced by it, resulting in a blue individual. The interaction described by the Incomplete Dominance model results in three different phenotypes. In this case, an individual can be red, blue or purple. Both alleles affect the phenotype and are noted with capital letters (R and B).

The Incomplete Dominance inheritance is an alternative to genotype to phenotype dominance schemes (Melita and Holban, 2016b) in genetic algorithms and is the approach adopted in our algorithm. Moreover, this method does not require an explicit scheme for genotype-phenotype mapping. The fitness of each individual is consequently evaluated as the mean accuracy of the two classifiers,

Chromosome No.	No. of features
1	9.17%
2	7.64%
3	5.81%
4	4.89%
5	5.19%
6	5.81%
7	5.50%
8	4.28%
9	4.28%
10	4.28%
11	6.11%
12	4.89%
13	2.44%
14	3.66%
15	3.66%
16	3.97%
17	4.89%
18	1.83%
19	5.19%
20	2.75%
21	1.22%
22	2.44%

Table 1: Default distribution of features on chromosomes.

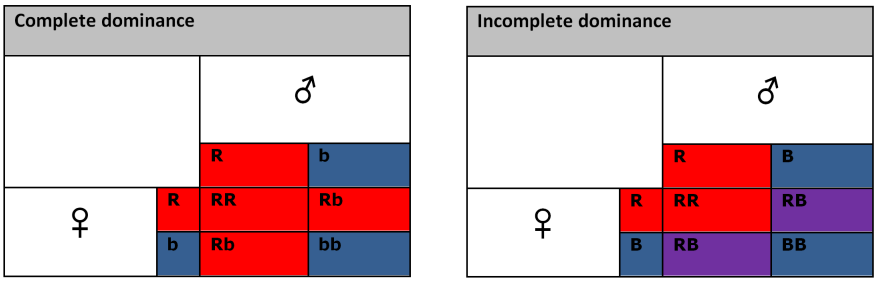

Figure 1: Complete vs. Incomplete Dominance.

each set of haploid chromosomes in the individual. With this approach it is possible to maintain a better variability in the late generations, as features from the poorly performing genotypes will be present in later generations when compared to the classical technique. Exploitation of the Incomplete Dominance model is optional. The value "ID2" for the parameter ID enables the Incomplete Dominance model, while the value "ID1" turns it off. In evaluating the fitness of an individual, any supervised classifier available in **MLInterfaces** package can be selected. The package offers a unified way to call supervised classifiers on data formatted according to "ExpressionSet class" specifications with several options (svmI, ldaI, rdaI, knnI, knn.cvI, randomForrestI, dldaI, nnetI, qdaI, naiveBayesI, etc). The cross-validation techniques implemented in **MLInterfaces** are also available for fitness evaluation in our package, and are accessible throught the trainTest parameter. The default value "LOG" for the trainTest parameter enables the leave-out-group cross-validation, while specifying "LOO", the user can opt for the leave-one-out cross-validation. Moreover, is if possible to shape the data into training and testing sets. A value of the form x:y for the trainTest parameter identifies the samples with indexes from x to y as training examples while all the others are used as testing set.

In the next step, the individuals are ranked according to their fitness. Our implementation offers the option of applying an elitist selection over the ranked individuals. The default value for elitism is NA, but a user can decide on the desired value for elitism, keeping the chosen number of best performing genotypes in the population.

Crossovers are applied next, between the two haploid sets of chromosomes in each individual. Two-point crossovers were preferred to the single-point alternative which preferentially affects the string ends. The two-point crossover follows the classical implementation. One two-point crossover is applied between homologous chromosomes, in each individual. A parameter to explicitly specify the chance for a crossover to occur is not implemented in our algorithm. The number of chromosomes implicitly affects the number of crossovers.

Another approach to crossover, modeled after the Random Assortment of Chromosomes in meiosis is also available. The Random Assortment of Chromosomes Crossover (Melita and Holban, 2016a) takes advantage of splitting the genomes in a number of chromosomes with variable size. When selected, two-point crossovers are performed between homologous chromosomes. After the crossovers are applied, the chromosomes are randomly assorted and distributed to one of the chromosomes set in each individual. This process, models the events that occur during meiosis I in eukaryotes. The user can choose at the initialization of the algorithm if the chromosomes will be randomly assorted through the randomAssortment parameter. The default value is TRUE. This operator is especially important when selecting a small number of features from a very large poll. In this case, the two-point crossover frequently recombines strings containing only zeros, with no effect on the very long genotypes. The Random Assortment operator offers a significant advantage in these situations. The distinctions between these approaches to recombination are highlighted in Figure 2, a part of the R output using with the code:

```
> library(dGAselID)
> set.seed(1357)
> c1<-rep(0, 10)
> c2<-rep(1, 10)
> individual<-rbind(c1, c2)
> individual

> chrConf01<-rep(1, 10)
> chrConf01

> #Two-point crossover on genotypes with 1 chromosome
> Crossover(individual [1, ], individual [2, ], chrConf01)

> chrConf03<-c(rep(1, 4), rep(2, 3), rep(3, 3))
> chrConf03

> #Two-point crossovers on genotypes with 3 chromosomes
> cr3<-Crossover(individual [1, ], individual [2, ], chrConf03)
> cr3

> #Random Assortment on the recombined genotypes with 3 chromosomes
> RandomAssortment(cr3, chrConf03)
```

Subsequently, the haploid sets of chromosomes with a higher fitness are kept and the others are discarded from each individual and mutations are applied with a chance that is specified by the user when initializing the algorithm. These haploid sets and the genotypes obtained thru crossovers are

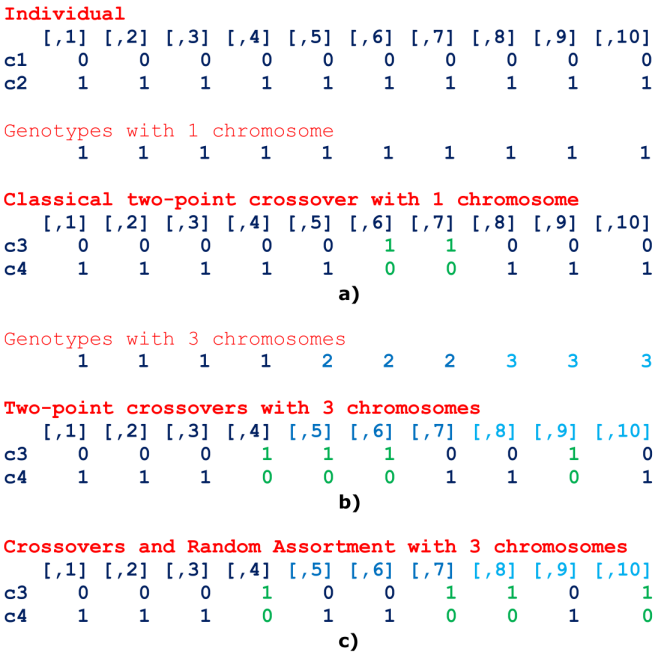

Figure 2: Recombination operators a) classical two-point crossover with 1 chromosome, b) two-point crossover with 3 chromosomes, c) two-point crossover with 3 chromosomes and Random Assortment

then assembled and a new generation of randomly generated individuals is created. Six different operators for mutation are available in the package. They can be used solitarily or in any combination, to support exploration with the genetic algorithm. The classical point mutation is implemented. However, when selecting a small number of features from a large poll with a genetic algorithm, the point mutation has the tendency to progressively increase the number of activated genes in genotypes, with every generation. To address this drawback, we implemented several mutation mechanisms inspired from genetics. These operators take advantage of the genotypes being partitioned on different chromosomes. The alternatives to point mutation available in the **dGAselID** package are:

1. Nonsense Mutation,
2. Frameshift Mutation,
3. Large Segment Deletion,
4. Whole Chromosome Deletion,
5. Transposons.

The Nonsense Mutation operator annuls all the genes on a chromosome following a randomly selected locus (treated as a stop codon). The other chromosomes in the genotype are not influenced by the nonsense mutation. **The Frameshift Mutation operator** randomly selects a locus on an arbitrarily selected chromosome. The gene at the selected locus is deleted and all the following chain is shifted with one position to the left. The last locus on the implicated chromosome is subsequently annulled to conserve its length. Other chromosomes in the genotype are not altered by the mutation. **The Large Segment Deletion operator** annuls all the genes in a randomly generated interval on an arbitrarily selected chromosome. **The Whole Chromosome Deletion operator** acts in a similar fashion, but on the whole chromosome rather than an interval. **The Transposons operator** randomly selects a chromosome, a gene on that chromosome and a distance for relocation. The elected gene is then transferred at a locus indicated by the generated distance. All the mutation operators occur with chances specified by designated parameters. The mutated genotypes are verified to have at least 4 active genes and the invalid mutations are not inherited, to prevent errors during the subsequent fitness evaluations. The following code demonstrates the mutation operators. The R output is partially presented in Figure 3.

```
> library(ALL)
> data(ALL)
>
> demoALL<-ALL[1:12, 1:8]
>
> set.seed(1234)
```

```
> population<-InitialPopulation(demoALL, 4, 9)
> individuals<-Individuals(population)
> individuals
>
> set.seed(123)
> pointMutation(individuals, 4)
>
> chrConf<-splitChromosomes(demoALL, 2)
> chrConf
> individuals
>
> set.seed(123)
> nonSenseMutation(individuals, chrConf, 20)
>
> set.seed(123)
> frameShiftMutation(individuals, chrConf, 20)
>
> set.seed(123)
> largeSegmentDeletion(individuals, chrConf, 20)
>
> set.seed(123)
> wholeChromosomeDeletion(individuals, chrConf, 20)
>
> set.seed(123)
> transposon(individuals, chrConf, 20)
```

```
> individuals
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          1          0       1       1       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> set.seed(123)
> pointMutation(individuals, 4)
       Applying 1 Point Mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          1          1       1       1       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> chrConf<-splitChromosomes(demoALL, 2)
> chrConf
 [1] 1 1 1 1 1 1 2 2 2 2 2 2
>
> set.seed(123)
> nonSenseMutation(individuals, chrConf, 20)
       Applying 1 NonSenseMutation mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          0          0       0       0       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> set.seed(123)
> frameShiftMutation(individuals, chrConf, 20)
       Applying 1 FrameShiftMutation mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          1          0       1       1       0          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> set.seed(123)
> largeSegmentDeletion(individuals, chrConf, 20)
       Applying 1 LargeSegmentDeletion mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          0          0       0       1       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> set.seed(123)
> wholeChromosomeDeletion(individuals, chrConf, 20)
       Applying 1 WholeChromosomeDeletion mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       0       0          0          0       0       0       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
>
> set.seed(123)
> transposon(individuals, chrConf, 20)
       Applying 1 Transposons mutations...
  Id 1000_at 1001_at 1002_f_at 1003_s_at 1004_at 1005_at 1006_at 1007_s_at 1008_f_at 1009_at 100_g_at 1010_at
1  1       0       1          1          1       1       0       1          1          0       1         1        1
2  1       1       1          1          1       0       1       1          1          0       0         1        1
3  2       1       1          1          1       0       1       1          1          1       0         1        0
4  2       1       0          1          1       1       1       1          0          1       1         0        1
```

Figure 3: Partial R output illustrating the mutation operators

Another iteration on the new generation follows. The number of generations is established at the initialization of the algorithm.

Argument	Description
x	The dataset in "ExpressionSet class" format
response	The response variable
method	Supervised classifier for fitness evaluation
trainTest	Specifies the training set or the cross-validation method
startGenes	Number of alleles=1 in the starting genomes
populationSize	Initial populations size
iterations	Number of generations
noChr	The number of desired chromosomes
elitism	Elitism in percentages
ID	Dominance
pMutationChance	Chance for a Point Mutation to occur
nSMutationChance	Chance for a Non-sense Mutation to occur
fSMutationChance	Chance for a Frameshift Mutation to occur
lSDeletionChance	Chance for a Large Segment Deletion to occur
wChrDeletionChance	Chance for a Whole Chromosome Deletion to occur
transposonChance	Chance for a Transposon Mutation to occur
randomAssortment	Random Assortment of Chromosomes for recombinations
embryonicSelection	Remove chromosomes with fitness < specified value
EveryGeneInInitialPopulation	Request for every gene to be present in the initial population
nnetSize	For nnetI
nnetDecay	For nnetI
rdaAlpha	For rdaI
rdaDelta	For rdaI

Table 2: Parameters accepted by the dGAselID() function.

Working with the dGAselID package

The **dGAselID** package is structured around the `dGAselID()` function. This function manages the initial parameters for the algorithm and, depending on the user selected options, sets the stage for the experiment. The `dGAselID()` function calls other functions for the different steps and options in the algorithm. The arguments accepted by `dGAselID()` along with short descriptions are presented in Table 2. The other functions, for different operators used during the genetic algorithm search, are summarized in Table 3.

Graphical representations of the evolution are available with the built-in functions in real-time. The maximum and average accuracy, accompanied by the most frequently selected genes can be displayed after each generation, offering a very intuitive image of the evolution. Evidence about the number of individuals in the current population, crossovers or the number of mutations are displayed for each generation.

The algorithm retains various data about the evolution for further analysis. For each gene, the frequency of selection across generations is recorded, along with other characteristics of the evolution. The output data format with a hypothetical result is shown below, together with a description of the recorded variables in the Table 4.

```
> ## Not run:
> names(result)        #hypothetical result
[1] "DGenes"          "dGenes"          "MaximumAccuracy" "MeanAccuracy"
[5] "MinAccuracy"     "BestIndividuals"
> ## End(Not run)
```

Example

We illustrate the functionality of the **dGAselID** package with the **ALL** dataset (Li, 2009), a very known set of real DNA microarray data, available in Bioconductor. We are searching for the differentially expressed genes that could characterize the patients suffering from acute lymphoblastic leukemia but have different BCR/ABL classification, negative or positive. For this example, we use a subset of the original ALL data, non-specifically and specifically filtered to 628 features and 79 samples, from the

Function	Description
dGAselID()	Main function
AnalyzeResults()	Ranks individuals according to their fitness and records the results
Crossover()	Operator for the two-point crossover
Elitism()	Performs elitism for the desired threshold
EmbryonicSelection()	Deletes individuals with a fitness below a specified threshold
EvaluationFunction()	Evaluates the individuals' finesses after each iteration
frameShiftMutation()	Operator for the frameshift mutation
Individuals()	Generates individuals from haploid chromosome sets
InitialPopulation()	Generates the initial random haploid chromosome sets
largeSegmentDeletion()	Operator for the large segment deletion mutation
nonSenseMutation()	Operator for the nonsense mutation
PlotGenAlg()	Plots the evolution after each generation
pointMutation()	Operator for the point mutation
RandomAssortment()	Performs the Random Assortment of chromosomes
RandomizePop()	Creates the random population for the next generation
splitChromosomes()	Divides the genotypes in a set with the desired number of chromosomes
transposon()	Operator for the transposon mutation
wholeChromosomeDeletion()	Operator for the whole chromosome deletion mutation

Table 3: Functions in the dGAselID package.

Variable	Description
DGenes	The occurrences in selected genotypes for every gene
dGenes	The occurrences in discarded genotypes for every gene
MaximumAccuracy	Maximum accuracy in every generation
MeanAccuracy	Average accuracy in every generation
MinAccuracy	Minimum accuracy in every generation
BestIndividuals	Best individual in every generation

Table 4: dGAselID() output.

12625 features and 128 patients in the complete data set. The data was filtered using the capabilities offered in the **genefilter** package (Gentleman et al., 2016). The algorithm works as well on the original data set, but the filtered data is searched faster. However, our experiments with real data show that more reliable results are obtained with full featured data.

The code for constructing the dataset used in the following examples is presented below:

```
> library(genefilter)
> library(ALL)
> data(ALL)
> bALL = ALL[, substr(ALL$BT,1,1) == "B"]
> smallALL = bALL[, bALL$mol.biol %in% c("BCR/ABL", "NEG")]
> smallALL$mol.biol = factor(smallALL$mol.biol)
> smallALL$BT = factor(smallALL$BT)
> f1 <- pOverA(0.25, log2(100))
> f2 <- function(x) (IQR(x) > 0.5)
> f3 <- ttest(smallALL$mol.biol, p = 0.1)
> ff <- filterfun(f1, f2, f3)
> selectedsmallALL <- genefilter(exprs(smallALL), ff)
> smallALL = smallALL[selectedsmallALL, ]
```

An example of function call is:

```
> set.seed(149)
> resNoID<-dGAselID(smallALL, "mol.biol", trainTest = 1:79, startGenes = 12,
+ populationSize = 200,  iterations = 300, noChr = 5, pMutationChance = 0.0075,
+ elitism = 4)
```

The choice for evaluation function was `knn.cvI` from the **MLInterfaces** package, with the parameters `k=3` and `l=2`. This is the default `method` in the package. The evaluation function used in this example, `knn.cvI`, is a kNN classifier with the leave-one-out cross validation embedded. For this reason, the requirement for the `trainTest` parameter is special, addresses all the instances in the data, `1:79`. For any other supervised classifier, the `trainTest` parameter shapes the training and testing subsets in the same fashion as the `trainInd` parameter in MLInterfaces. When cross-validation is required, the `trainTest` parameter specifies the desired method as `"LOO"` or `"LOG"`, and is equivalent to `xvalSpec("LOO")` or `xvalSpec("LOG")` respectively, in MLInterfaces package. An illustration of the evolution is presented in Figure 4. The figure pictures a juncture during the search, including the information provided by the verbose mode and graphical representations of the evolution, as they appear in real-time on the computer screen. The evolutions of the Maximum Accuracy, Average Accuracy and the most frequently selected genes are presented after each generation.

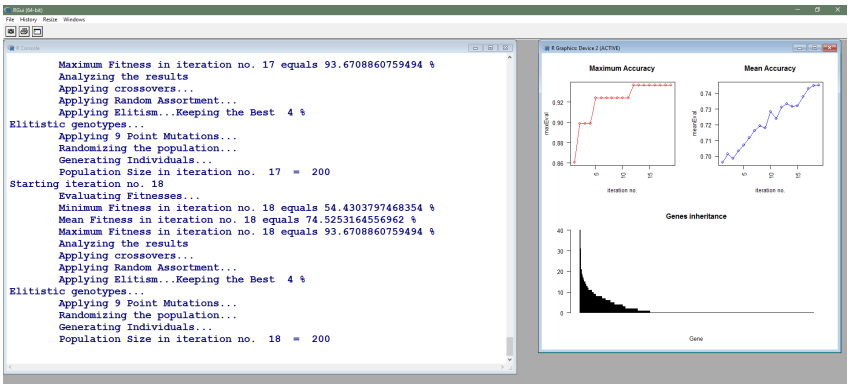

Figure 4: Screenshot of the algorithm execution after 18 generations.

After the desired number of generations, the evolution of the Maximum Accuracy and Average Accuracy in every generation and the most frequently selected genes can be displayed with the included functions. The Figure 5 represents the most frequently selected genes after the specified number of generations, 300 in our example. Their characteristics can be acquired with methods already implemented in Bioconductor. The 10 most selected genes are presented and could be obtained using **hgu95av2.db** (Carlson, 2016) with the code below. The selected genes can be studied and evaluated with all the methods available in Bioconductor. For reliable and interpretable results, an adequate number of replications are mandatory in such an experiment, due to the stochastic makeup of the genetic algorithms. The evolution of the maximum accuracy and the average accuracy can be plotted as illustrated in the Figure 6 and Figure 7, respectively.

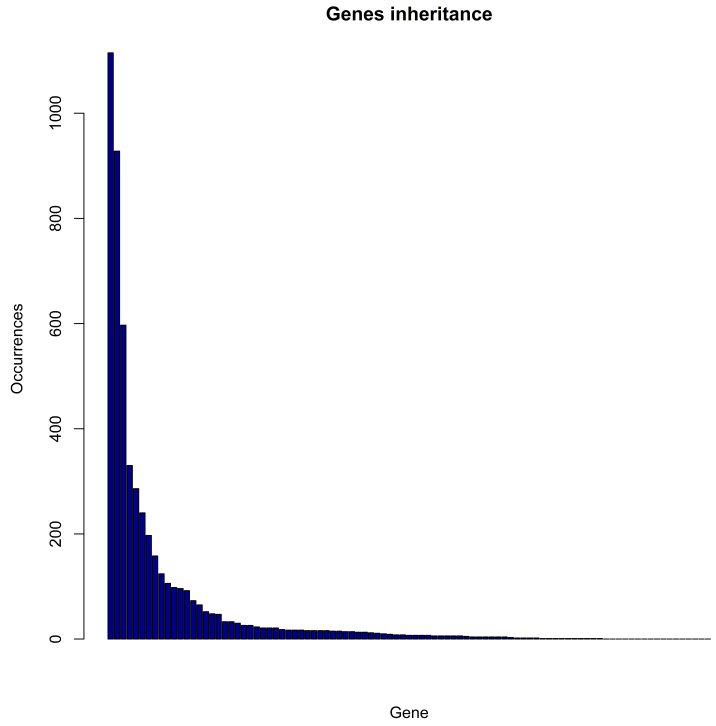

Figure 5: The most frequently selected genes after 300 generations.

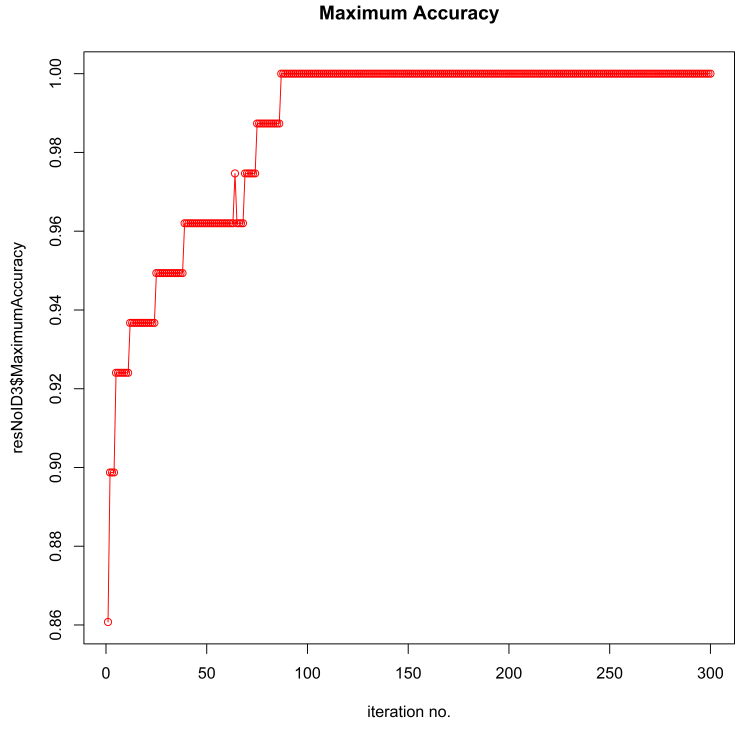

Figure 6: Evolution of the maximum accuracy after 300 generations.

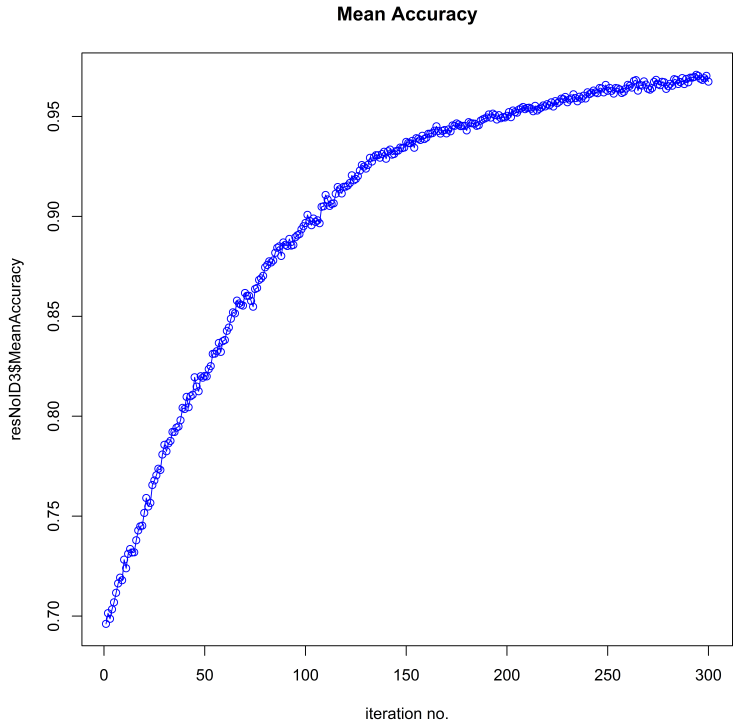

Figure 7: Evolution of the average accuracy after 300 generations.

```
> library(hgu95av2.db)
> DGenes<-resNoID$DGenes
> selectedD<-colnames(DGenes)[order(DGenes, decreasing = TRUE)]
> mget(selectedD, hgu95av2GENENAME, ifnotfound=NA)[1:10]
$`1635_at`
[1] "ABL proto-oncogene 1, non-receptor tyrosine kinase"

$`39730_at`
[1] "ABL proto-oncogene 1, non-receptor tyrosine kinase"

$`39070_at`
[1] "fascin actin-bundling protein 1"

$`34362_at`
[1] "solute carrier family 2 member 5"

$`39338_at`
[1] "S100 calcium binding protein A10"

$`40091_at`
[1] "B-cell CLL/lymphoma 6"

$`1135_at`
[1] "G protein-coupled receptor kinase 5"

$`38385_at`
[1] "destrin, actin depolymerizing factor"

$`40396_at`
[1] "purinergic receptor P2X 5"

$`38069_at`
[1] "chloride voltage-gated channel 7"
```

Other graphical representation tools offered in R are readily available for further investigation. An image of the evolution for the best individual in every generation can be depicted as in Figure 8. The tendency to increment the number of active genes in the genotypes with the successive generations, induced by the point mutation, becomes apparent. In contrast, the transposon mutation does not convey this inconvenience, as illustrated in Figure 9. The graphical representations are accessible with the code:

```
> bestsNoID<-resNoID$BestIndividuals
> dev.off()
> image(1:ncol(bestsNoID), 1:nrow(bestsNoID), t(bestsNoID),
+ xlim = c(0, ncol(bestsNoID)), ylim = c(0, nrow(bestsNoID)), col = c("white", "red"),
+ cex.axis = 0.7, cex.lab = 0.8, cex.main = 1.2, lty = 1, lwd = 2, las= 2, xaxs = "r",
+ yaxs = "r", pty = "m", ylab = "Generation no.", xlab = "Gene no.",
+ main = "Best Individuals with the Point Mutation")

> set.seed(149)
> restransp<-dGAselID(smallALL, "mol.biol", trainTest = 1:79, startGenes = 12,
+ populationSize = 200, iterations = 300, noChr = 5, pMutationChance = 0,
+ transposonChance = 2, elitism= 4)

> beststransp<-restransp$BestIndividuals
> dev.off()
> image(1:ncol(beststransp), 1:nrow(beststransp), t(beststransp),
+ xlim = c(0, ncol(beststransp)), ylim = c(0, nrow(beststransp)),
+ col = c("white", "red"), cex.axis = 0.7, cex.lab = 0.8, cex.main = 1.2, lty = 1,
+ lwd = 2, las = 2, xaxs = "r", yaxs = "r", pty = "m", ylab = "Generation no.",
+ xlab = "Gene no.", main = "Best Individuals with the Transposon Mutation")
```

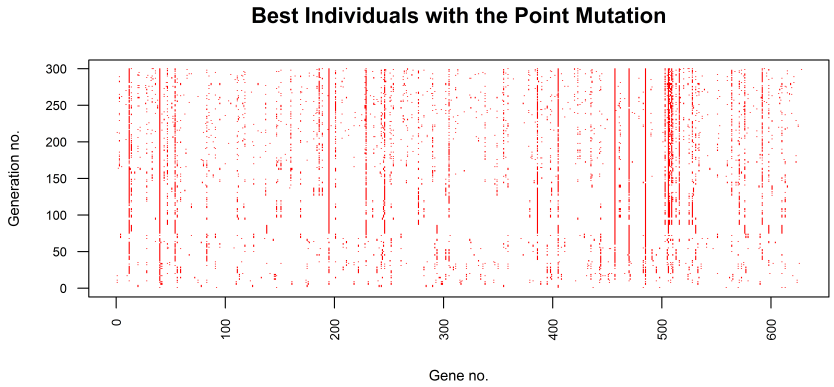

Figure 8: Evolution of the best individual after 300 generations with Point Mutation.

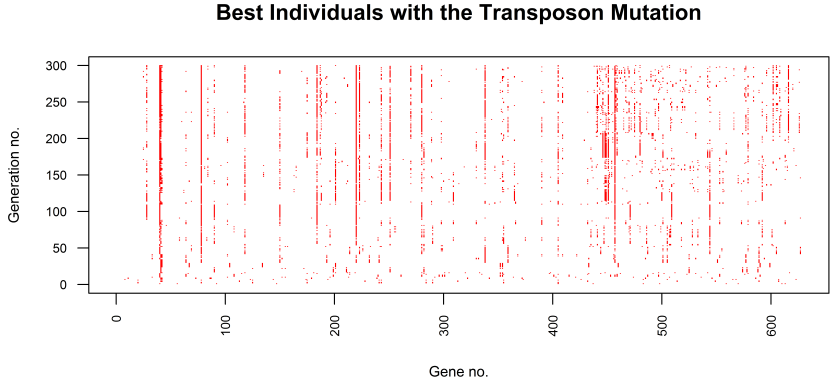

Figure 9: Evolution of the best individual after 300 generations with Transposon Mutation.

An illustrative comparison between the classical and the Incomplete Dominance implementations is accessible with the subsequent code. The two approaches can be assessed in terms of evolutions of the maximum and average fitness, as depicted in Figure 10 and Figure 11, respectively. The **scales** package (Wickham, 2016) is very useful for the direct visual comparison. The most frequently selected features are compared in Figure 12. It is noticeable that the Incomplete Dominance approach favors exploration, while still evolving very solidly. This behavior is desirable when selecting features with a genetic algorithm. The same tendency is perceptible when examining Figure 12, where the number of significant features is higher with the Incomplete Dominance method. Multiple replications of an experiment are mandatory to draw reliable conclusions. This example is presented for illustration purpose only.

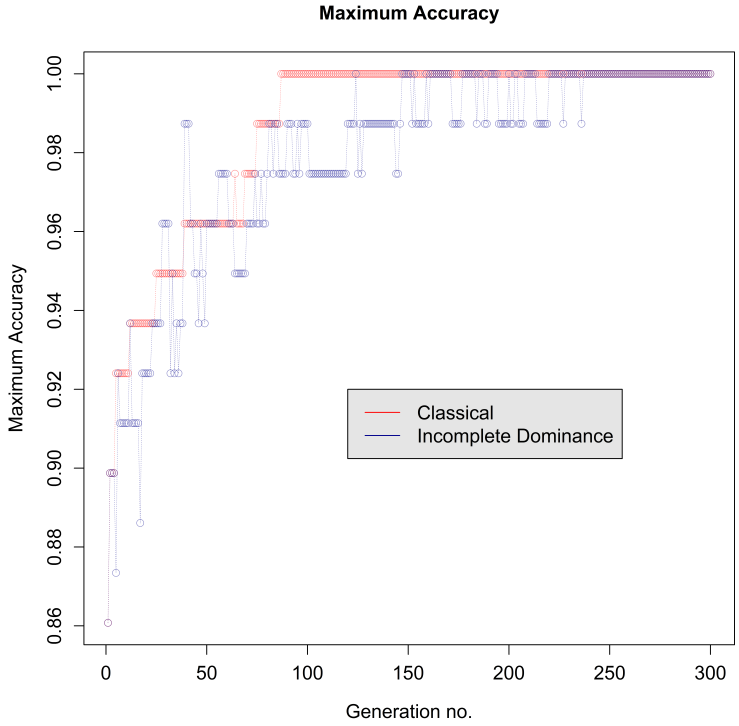

Figure 10: Comparative evolution of the maximum fitness over 300 generations.

```
> library(scales)
> set.seed(149)
> resID<-dGAselID(smallALL, "mol.biol", trainTest = 1:79, startGenes = 12,
+ populationSize = 200, iterations = 300, noChr = 5, pMutationChance = 0.0075,
+ elitism = 4, ID = "ID2")

> dev.off()
> par("xlog"=FALSE)
> plot(resNoID$MaximumAccuracy, type = "o", col = alpha("red", 0.5), pch = 1,
+ cex.axis = 1.2, cex.lab = 1.2, cex.main = 1.2, lty = 3, lwd = 0.5,
+ xlab = "Generation no.", ylab = "Maximum Accuracy", main = "Maximum Accuracy")
> points(resID$MaximumAccuracy, type = "o", col = alpha("darkblue", 0.5), pch = 1,
+ lty = 3, lwd = 0.5)
> legend(120, 0.92, c("Classical", "Incomplete Dominance"), cex = 1.2,
+ col = c("red", "darkblue"), merge = FALSE, bg = "gray90", lty = c(1, 1, 1))

> plot(resNoID$MeanAccuracy, type = "o", col = alpha("red", 0.5), pch = 1, lwd = 0.5,
+ cex.axis = 1.2, cex.lab = 1.2, cex.main = 1.2, xlab = "Generation no.",
+ ylab = "Mean Accuracy", main = "Mean Accuracy")
> points(resID$MeanAccuracy, type = "o", col = alpha("darkblue", 0.5), pch = 1,
+ lty = 3, lwd = 0.5)
> legend(120, 0.8, c("Classical", "Incomplete Dominance"), cex = 1.2,
+ col = c("red", "darkblue"), merge = FALSE, bg = "gray90", lty = c(1, 1, 1))
```

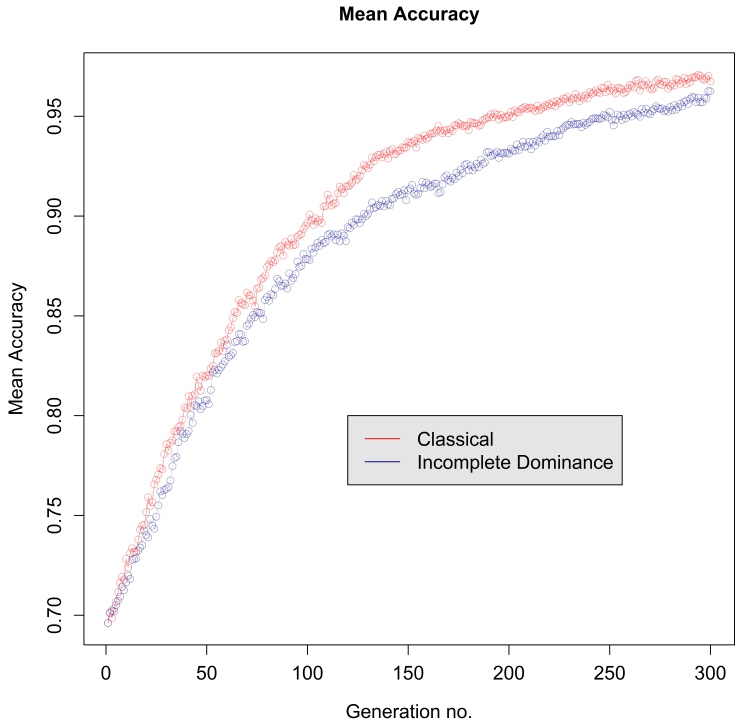

Figure 11: Comparative evolution of the average fitness over 300 generations.

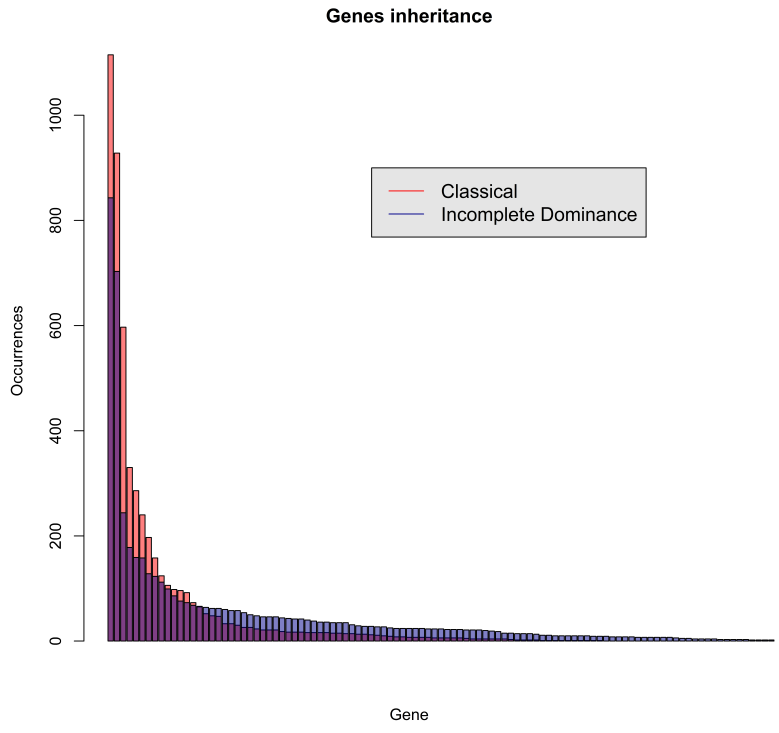

Figure 12: Comparatison of the most selected genes after 300 generations.

```
> DGenes1<-resNoID$DGenes[sort(resNoID$DGenes, decreasing = TRUE,
+ index.return = TRUE)$ix]
> DGenes2<-resID$DGenes[sort(resID$DGenes, decreasing = TRUE, index.return = TRUE
+ )$ix]
> significant1<-DGenes1[sort(resNoID$DGenes, decreasing = TRUE, index.return = TRUE
+ )$ix] > floor(5*max(DGenes1)/100)
> significant2<-DGenes2[sort(resID$DGenes, decreasing = TRUE, index.return = TRUE
+ )$ix] > floor(5*max(DGenes2)/100)
> barplot(DGenes1[significant1], main = "Genes inheritance", xlab = "Gene",
+ ylab = "Occurrences", col = alpha("red", 0.5), beside = FALSE, add = FALSE)
> barplot(DGenes2[significant2], main = "Genes inheritance", xlab = "Gene",
+ ylab = "Occurrences", col = alpha("darkblue", 0.5), beside = FALSE, add = TRUE)
> legend(50, 900, c("Classical", "Incomplete Dominance"), cex = 1.2, col = c("red",
+ "darkblue"), merge = FALSE, bg = "gray90", lty = c(1, 1, 1))
```

Conclusions

The **dGAselID** package provides a creative approach to feature selection in high dimensional data. The package utilizes the data format used in Bioconductor and is readily operational for microarray data analysis. The algorithm is flexible for high dimensional data other than microarray, when data is provided according to the `"ExpressionSet class"` specifications. In our experience, the diploid implementation offers advantages over the haploid GA, especially when cross-validation techniques are engaged. The Incomplete Dominance approach allows for a diploid framework, bypassing the requirement to specify a dominance scheme. Also, the Incomplete Dominance inheritance models an evolution process present in nature, tested by billions of years of evolution. Moreover, the diploid structure is a foundation for crossover and mutation operators that model the natural processes and provide improvements in performance over the classical versions. In our tests, the Random Assortment of Chromosomes provides an important performance advantage, in many situations.

Future development

The main disadvantages of the algorithm presented in **dGAselID** package are the necessity to replicate an experiment several times for reliable results, given the fortuity in generating the initial population, and the tendency of the GA to converge in local optima. Following the conduit in evolutionary computation, we will reconsider the principles of evolution and accurately model operators for crossover and mutation to offer a better tension between exploration and exploitation.

Bibliography

D. Akdemir, J. Sanchez, and J.-L. Jannink. Optimization of genomic selection training populations with a genetic algorithm. *Genetics Selection Evolution*, 47(1):38, 2015. URL https://doi.org/10.1186/s12711-015-0116-6. [p1]

J. Berard and A. Bienvenue. Sharp asymptotic results for simplified mutation-selection algorithms. *The Annals of Applied Probability*, 13(4):1534–1568, 2003. URL https://doi.org/10.1214/aoap/1069786510. [p1]

V. Carey, R. Gentleman, J. Mar, J. Vertrees, and L. Gatto. *MLInterfaces: Uniform interfaces to R machine learning procedures for data in Bioconductor containers*, 2016. URL http://bioconductor.org/packages/MLInterfaces/. R package version 1.52.0. [p2]

M. Carlson. *hgu95av2.db: Affymetrix Human Genome U95 Set annotation data (chip hgu95av2)*, 2016. URL http://bioconductor.org/packages/hgu95av2.db/. R package version 3.2.3. [p9]

B. Doerr and C. Doerr. A tight runtime analysis of the (1 + (λ,λ)) genetic algorithm on onemax. *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1423–1430, 2015. URL http://doi.acm.org/10.1145/2739480.2754683. [p1]

S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, Apr. 2002. ISSN 0304-3975. URL https://doi.org/10.1016/S0304-3975(01)00182-7. [p1]

R. Gentleman, V. Carey, W. Huber, and F. Hahne. *genefilter: genefilter: methods for filtering genes from high-throughput experiments*, 2016. URL https://bioconductor.org/packages/genefilter/. R package version 1.56.0. [p9]

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675. [p1]

J. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975. ISBN 0472084607. Extended new Edition, MIT Press, Cambridge (1992). [p1]

W. Huber, V. Carey, R. Gentleman, ..., and M. Morgan. Orchestrating high-throughput genomic analysis with bioconductor. *Nature Methods*, 12(2):115–121, 2015. ISSN 1548-7091. URL https://doi.org/10.1038/nmeth.3252. [p1]

D. Kepplinger. *gaselect: Genetic Algorithm (GA) for Variable Selection from High-Dimensional Data*, 2015. URL https://cloud.r-project.org/web/packages/gaselect/index.html. R package version 1.0.5. [p2]

X. Li. *ALL: A data package*, 2009. URL http://bioconductor.org/packages/release/data/experiment/html/ALL.html. R package version 1.14.0. [p7]

N. Melita and S. Holban. The random assortment for selecting features with genetic algorithms in microarray data analysis. *in publishing*, 2016a. [p4]

N. Melita and S. Holban. An incomplete dominance genetic algorithm approach to microarray data analysis. *2016 IEEE 12th International Conference on Intelligent Computer Communication and Processing (ICCP)*, 2016b. URL https://doi.org/10.1109/ICCP.2016.7737137. [p2]

N. Melita, I. Popescu, and S. Holban. A genetic algorithm approach to dna microarrays analysis of pancreatic cancer. *Advances in Electrical and Computer Engineering*, 8(2):43–48, 2008. ISSN 1582-7445. URL https://doi.org/10.4316/AECE.2008.02008. [p2]

M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262631857. [p1]

T. Pajala. *mogavs: Multiobjective Genetic Algorithm for Variable Selection in Regression*, 2016. URL https://cloud.r-project.org/web/packages/mogavs/index.html. R package version 1.0.1. [p2]

G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Trans. Neural Net.*, 5(1): 96–101, 1994. URL https://doi.org/10.1109/72.265964. [p1]

L. Scrucca. On some extensions to ga package: hybrid optimisation, parallelisation and islands evolution. *Submitted to R Journal*, 2016. URL http://arxiv.org/abs/1605.01931. [p1]

F. Tenorio. *gaoptim: Genetic Algorithm optimization for real-based and permutation-based problems*, 2013. URL https://cloud.r-project.org/web/packages/gaoptim/index.htmll. R package version 1.1. [p1]

C.-S. Tsou. *nsga2R: Elitist Non-dominated Sorting Genetic Algorithm based on R*, 2015. URL https://cloud.r-project.org/web/packages/nsga2R/index.html. R package version 1.0. [p1]

H. Wickham. *scales: Scale Functions for Visualization*, 2016. URL https://CRAN.R-project.org/package=scales. R package version 0.4.1. [p13]

E. Willighagen and M. Ballings. *genalg: R Based Genetic Algorithm*, 2015. URL https://cloud.r-project.org/web/packages/genalg/index.html. R package version 0.2.0. [p1]

M. Wolters. A genetic algorithm for selection of fixed-size subsets with application to design problems. *Journal of Statistical Software*, 68(1):1–18, 2015. URL https://doi.org/10.18637/jss.v068.c01. [p1]

B. Xue, M. Zhang, W. Browne, and X. Yao. Survey on evolutionary computation approaches to feature selection. *IEEE Transactions on Evolutionary Computation*, 20(4), 2016. URL https://doi.org/10.1109/TEVC.2015.2504420. [p1]

Nicolae Teodor MELITA
Department of Computer and Software Engineering
Politehnica University of Timisoara
Timisoara, RO-300223
Romania
nt_melita@yahoo.com

Stefan HOLBAN
Department of Computer and Software Engineering
Politehnica University of Timisoara
Timisoara, RO-300223
Romania
stefan.holban@cs.upt.ro