GALGO

An R package for Genetic Algorithm Searches
(Customized for Variable Selection in Functional Genomics)

Victor Trevino & Francesco Falciani
School of Biosciences
University of Birmingham
United Kingdom

November 2005,
February 2006

1   Overview

1.1   What is GALGO?

GALGO is a generic software package that uses **G**enetic **Alg**orithms to solve **O**ptimization problems involving the selection of variable subsets. In the current version, GALGO has a set of tools to support the development of statistical models from very large datasets, such as Genome wide functional data.

GALGO is implemented in the statistical programming environment R [1] using object-oriented programming under the S3 methods. We used S3 methods instead of S4 to benefit from R.oo package [2] whose main functionality is to pass objects parameters by reference instead of by value. The result is that the performance is drastically improved and the memory requirements are diminished. The R.oo package is therefore required for the functioning of GALGO.

GALGO can be used to solve any optimization problem, particularly in very large datasets where variable selection is an important issue. In this release GALGO include a series of methods to perform supervised classification. In order to use GALGO as a generic package an ad-hoc fitness function must be coded. Because of the large number of statistical functions available in R this is, in most cases, a straightforward operation and does not require extensive coding.

This manual describes the main functionality implemented in GALGO (v 1.08) and provides a step by step tutorial describing a typical application to biomarker discovery using gene expression profiling data. The manual also provides examples on how to implement ad-hoc fitness function for the solution of regression problems.

1.2   Why developing GALGO?

In the analysis of large datasets, such as data obtained using Functional Genomics Technologies, the selection of gene signatures predictive of sample features (for example disease type) is a difficult problem. Commonly the number of samples is very low (hundreds or dozens) and certain aspects of the samples are known (for example disease type, strain, treatment, etc). One of the most basic problems is the selection of genes whose profile is, in some way, associated to the known sample type, which in turn would allow acquiring more knowledge about the mechanism of action, generating new hypothesis, directing further research, selecting biomarkers, and choosing

potential drug targets. In statistics, this association of profiles to known sample types is called "supervised classification" and there are several classification methods that "test" if genes are related to samples phenotype. These methods can be subdivided in univariate and multivariate methods. Univariate methods evaluate each variable (e.g. a gene) at the time for its ability to discriminate between two or more groups of samples. PAMR (Tibshirani *et al.* 2002), TNASAS (Vaquerizas *et al.*, 2005), and GeneSpring (Silicon Genetics, Redwood City, CA) are perhaps the most software applications used by the Functional Genomics community that implement univariate variable selection methods for classification. These tools use different statistics to identify genes that are differentially expressed between two or more groups of samples and then uses the most differentially expressed to construct a statistical model (Figure 1). These methods have demonstrated to perform well, however, in some cases they can be ineffective regardless of the classification method used. An obvious conceptual limitation of univariate approaches is also the lack of consideration that genes works in the contexts of interconnected pathways and therefore it is their behaviour as a group that may be predictive of the phenotypic variables. Multivariate selection methods may seem to be more suitable for the analysis of Biological data since variables (such as gene expression values) are tested in combination to identify interactions between genes. However, the extremely large number of models that can be constructed from different combination of thousands of genes cannot be extensively evaluated using available computational resources. An alternative to the extensive analysis of all possible models is the use of search procedures that "explore" the data looking for good, although not optimal, sets of variables. Recently, Markov Chain Monte Carlo methods and Genetic algorithms have been applied successfully to the analysis of microarray data (Li *et al.* 2001; Ooi *et al.* 2003; Sha *et al.* 2004).

At present, there is no available software package to support the development of statistical models using multivariate variable selection strategies. To address this issue we have developed GALGO, an R package that uses a genetic algorithm search procedure coupled to statistical modelling methods for supervised classification (Figure 2). GALGO is relatively easy to use, can manage parallel searches and has a toolset for the analysis of models. Although GALGO include a number of statistical modelling methodologies to solve classification problems, GALGO can be used as a general tool to solve optimization problems. This requires rewriting the fitness function to specify the criteria for the selection of good variable subset. Because of the functionality that is already available in R, this can be achieved relatively easily. This manual provides a step-by-step tutorial to solve classification

problems using microarray data. It also provides examples of the use of GALGO as a general tool to solve optimization problems.

UNIVARIATE VARIABLE SELECTION
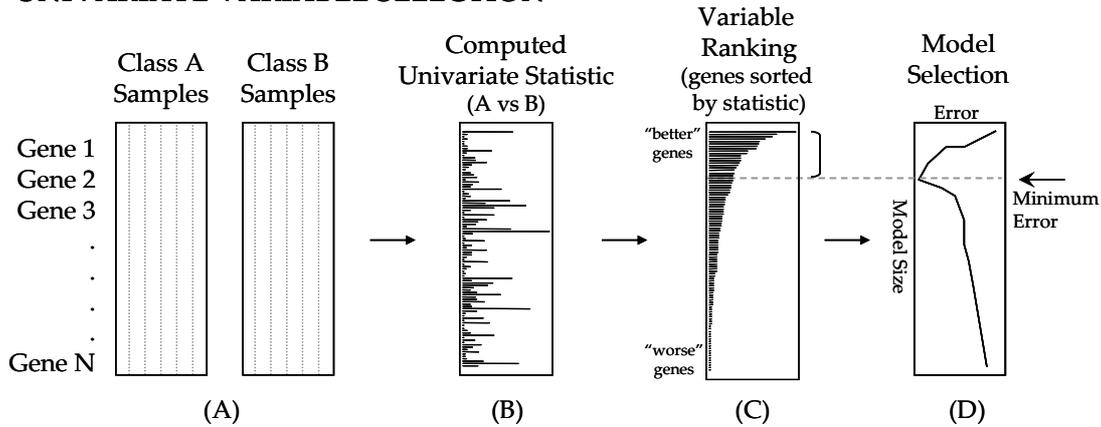
Figure 1 Schematic representation of univariate variable selection. A dataset of two classes of samples (A) is assessed using a univariate test (B) to rank genes by their sole ability to distinguish between classes (C). Then, a forward selection strategy using a classification method is used to detect the number of ranked genes that generates the lowest error (D).

MULTIVARIATE VARIABLE SELECTION IN GALGO
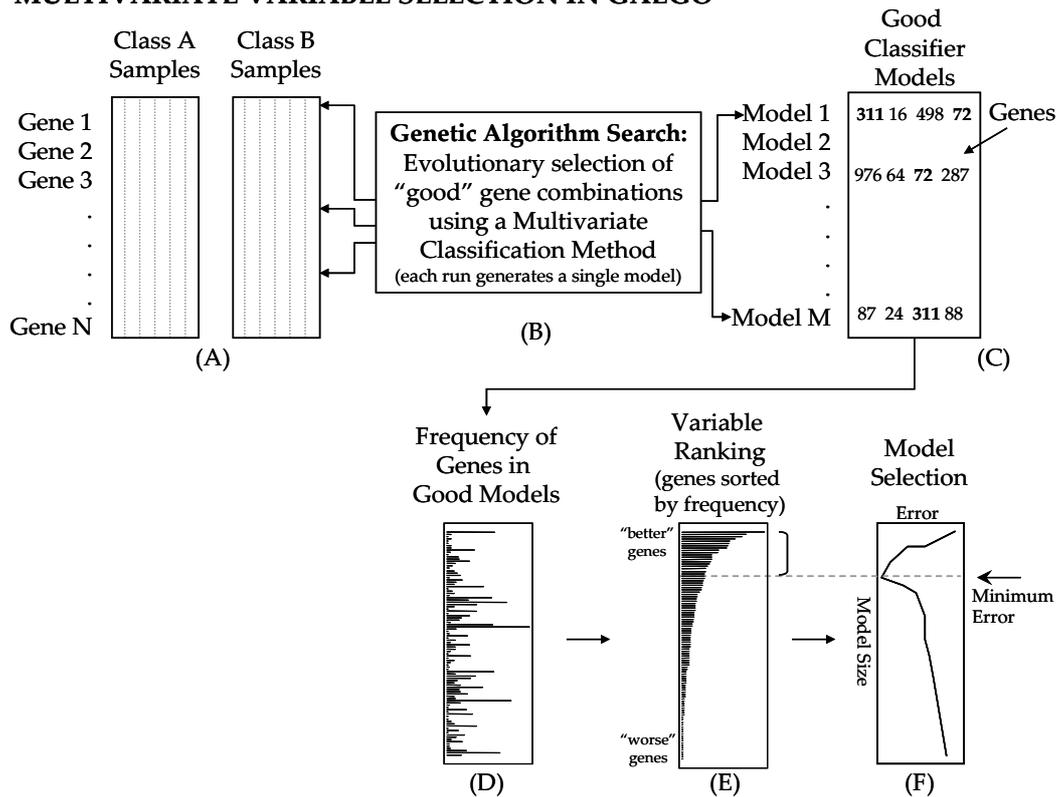
Figure 2 Schematic representation of multivariate variable selection. From a dataset of two classes of samples (A), a genetic algorithm (B) searches and evolves combination of genes (chromosomes representing a multivariate model) that distinguish between classes using a classification method. A number of models are generated performing this procedure several times (C). These models may differ in gene content but with similar high classification accuracy. Genes appearing multiple times in different models suggest these genes are important for the classification problem in a multivariate context. Therefore, the number of times (frequency) a gene appears in a model is computed (D). These frequencies

are used to rank genes (E). Then, a forward selection strategy is used to select a representative model that generates the lowest error (F).

1.3   *What is a Genetic Algorithm?*

Genetic Algorithms (GAs) are variable search procedures that are based on the principle of evolution by natural selection. The procedure works by evolving sets of variables (chromosomes) that fit certain criteria from an initial random population via cycles of differential replication, recombination and mutation of the fittest chromosomes. The concept of using *in-silico* evolution for the solution of optimization problems has been introduced by John Holland in 1975 (Holland 1975). Although their application has been reasonably widespread (see Goldberg's book (Goldberg, 1989)), they became very popular only when sufficiently powerful computers became available. What follows is a Step by Step description of the procedure in the context of a classification problem (see Figure 3 for a schematic representation of the procedure, note that we will use stages here to avoid confusion with those steps in the general GALGO pipeline):

Stage 1: The procedure initially creates a number of random variable sets (chromosomes). These variable sets form a population of chromosomes (niche).

Stage 2: Each chromosome in the population is evaluated for its ability to predict the group membership of each sample in the dataset (fitness function). This is achieved by training a statistical model. The GA tests the accuracy of the prediction and assigns a score to each chromosome that is proportional to the accuracy resulted in the fitness function.

Stage 3: When a chromosome has a score higher then a predefined value, this chromosome is selected and the procedure stops; otherwise, the procedure continues to stage 4.

Stage 4: The population of chromosomes is replicated. Chromosomes with a higher fitness score will generate a more numerous offspring.

Stage 5: The genetic information contained in the replicated parent chromosomes is combined through genetic crossover. Two randomly selected parent chromosomes are used to create two new chromosomes (Figure 4). This crossover mechanism allows a better exploration of possible solutions recombining good chromosomes.

Stage 6: Mutations are then introduced in the chromosome randomly. These mutations produce that new genes are used in chromosomes

Stage 7: The process is repeated from stage 2 until an accurate chromosome is obtained. The cycle of replication (stage 4), genetic cross-over (stage 5) and mutations (stage 6) is called generation.

Stage 1

Create Initial Population of Chromosomes with random genes

Artificial chromosome

Stage 2

Evaluate all chromosomes using the fitness function

Population and fitness value attached

Stage 3

if some fitness >= goal

yes

SELECT

no

Stage 4

Generate new popultation: Reproduce chromosomes proportionally to its fitness

new population

Stage 5

Random crossover between chromosomes pairs

crossover

Stage 6

Random mutatations on new population

mutation

Stage 7

Figure 3 Schematic representation of the GA Procedure.

crossover point

parent A

| Gene 7281 | Gene 542 | Gene 9567 | Gene 522 | Gene 892 |

parent B

| Gene 9823 | Gene 72 | Gene 95 | Gene 1827 | Gene 9432 |

crossover

parent A genes   parent B genes

| Gene 7281 | Gene 542 | Gene 9567 | Gene 1827 | Gene 9432 |

| Gene 9823 | Gene 72 | Gene 95 | Gene 522 | Gene 892 |

parent B genes   parent A genes
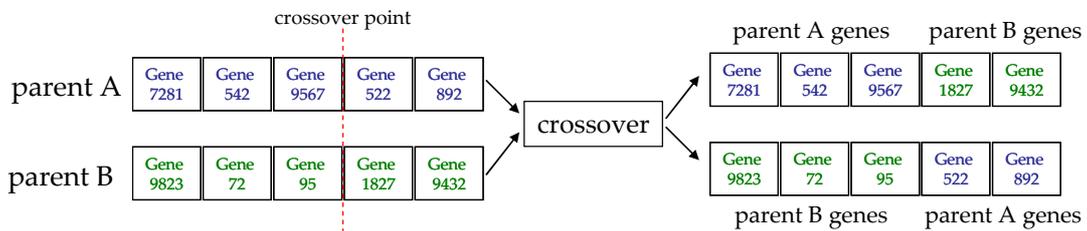
Figure 4 Schematic representation of the Crossover.

- DATA
- best chromosomes
- gene frequencies
- statistics

- best
- max
- statistics

- best
- max

- best
- max

- gene values

- shapes

| BigBang | ◁ | Galgo | ◁ | World | ◁ | Niche | ◁ | Chromosome | ◁ | Gene |

- blast
- accuracies
- model generation
- plots

- evaluate
- evolve
- plots

- evaluate
- progeny
- plots

- evaluate
- progeny
  - offspring
  - mutate
  - crossover
- plots

- mutate
- evaluate
- decode

- mutate

Fitness Function
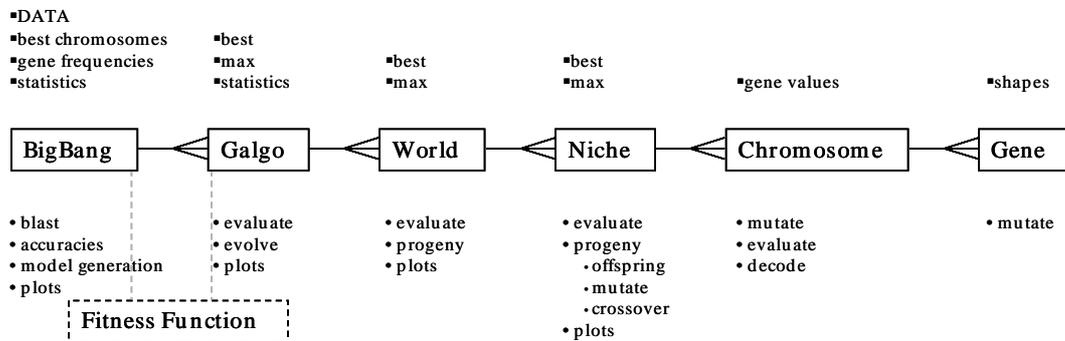
Figure 5 Simplified object-oriented structure of the GALGO package.

1.4 Implementation of Genetic Algorithms in GALGO package

We design GALGO package as an implementation of GA in object-oriented paradigm under the S3 methods in R using the objects created by R.oo package. Figure 5 shows an overview of the relationships of the GALGO objects. All objects can be extended and all methods can be overwritten. In the GA terminology variables are defined as *genes* whereas a subset of *n* variables that is assessed for its ability to fit a statistical model is called a *chromosome*. Populations of chromosomes are organized in *niches* that are independently evolving environments. However, niches have the possibility to occasionally exchange chromosomes with a process called *migration*. Multiple niches can then be part of a *world*. The *Galgo* object evolves a list of population objects (niches or worlds) and generates a *best* chromosome. The *BigBang* object collect these best chromosomes.

1.5 Statistical Model

A statistical model associates parameters computed from variables to make predictions or to study their relationships. In all models, it is necessary to make assumptions, which often facilitates the numerical solution but limit the applicability. For example, an assumption implicit in a t-test is that the values are normally distributed. To meet the assumptions, transformation of the data is sometimes necessary before the analysis. Models are generally based on mathematical relations and depend on both, parameters and data. The data is used to estimate the parameters of the model in a way that maximize the accuracy (or minimize the error). For instance, in a linear model *y=mx+b*, we usually know *x* and *y* whereas *m* and *b* are the unknown parameters of the model that are computed in such a way to minimize the error in the prediction of *y* given *x*. In this way, using the parameters *m* and *b* learned from known data, any new never seen *x* value can be evaluated into the model to predict its corresponding *y* value. Of course this prediction is subject to error, which would depend on the amount of data, data quality, values of the parameters, and on the model itself. In particular, the aim in classification models is the prediction of the class of unknown samples referred as *test cases* using the information of the known samples referred as *training cases* to estimate the parameters of the classification model. Different classification methods require particular parameters; however, here we are interested not in the parameters but in the prediction error made by models. That is, we are looking for models that produce small errors.

1.6 *Developing multivariate statistical models using GALGO: an overview of the Analysis pipeline*

Figure 6 summarize the analysis pipelines that can be built using GALGO. The first step is the specification of the data, the definition of the parameters for the GA search, the statistical model (classification method), and the error estimation. The step 2 consists on searching for gene combinations that are good classifiers. While the GA is exploring the space of variables for good solutions it is possible to visualize in real time the characteristics of the chromosome population in the course of one evolutionary event as well as the characteristics of the population of selected chromosomes. These plots are very useful diagnostics tools to understand the behaviour of the search. Once a large enough population of selected chromosomes is available, in step 3 we have several options for their analysis. The classification accuracy of the selected chromosomes can be established on the test data (this can be done on the original test data or as an average of several training-test data splits) and the results of this analysis can be summarised in a set of tables or plotted. In step 4, a single representative model of the chromosome population can be generated using a forward selection procedure that construct a model including the most frequent genes in the population of selected chromosomes in a new model (Li *et al.* 2001). The classification accuracy of this model can then be evaluated using the functions described before for the performance analysis of the population of selected chromosomes. Alternatively the population of selected chromosomes can be improved either by filtering chromosomes with a bad classification performance in the test data or by removing genes that do not contribute to the classification accuracy. Both individual and summary models can be visualized using heatmaps, PCA, sample profiles, or gene profiles.
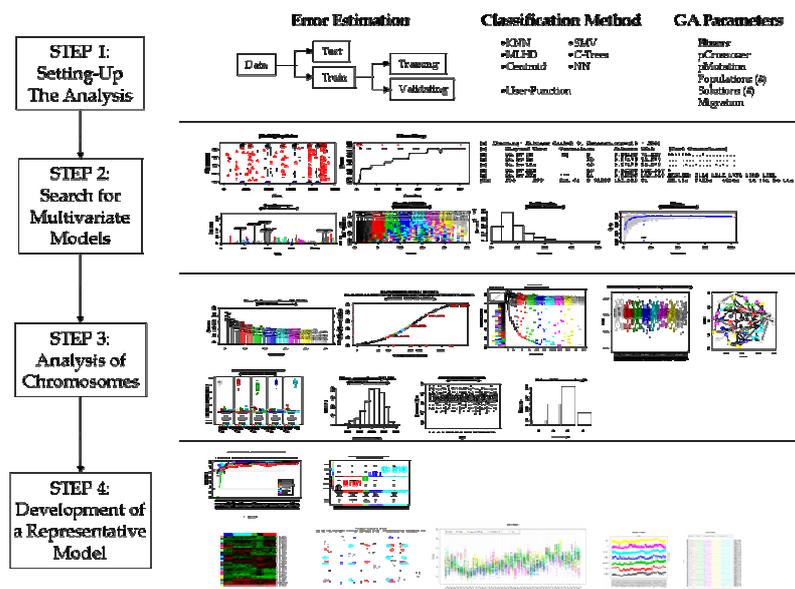

Figure 6 Overview of the analysis pipeline in GALGO package.

1.7 Comparison between Multivariate and Univariate Variable Selection Methods

We have previously reported the results of a comparison between two common univariate variable selection strategies (F-statistic and d-statistic) with the multivariate variable selection method implemented within GALGO in association with a number of classification methods (bioinformatics paper -submitted-). Models developed have been analysed in respect to classification accuracy, number of genes required to achieve the highest classification accuracy, and the identity of the genes selected in the models. In order to make sure that our comparison is of general validity we have used three different datasets. Table shown below is an extracted summary table for the dataset we will use in this tutorial.

ALL-Subclasses (5 Classes)			
	Model Selection	*Optimal Model*	
Gene Selection	*+Classifier*	*Size*	*Error*
F-statistic (univariate)	DLDA	200	0.02
F-statistic (univariate)	SVM	75	0.00
F-statistic (univariate)	RF	1000	0.17
F-statistic (univariate)	KNN	120	0.01
d-statistic (univariate)	PAM	439	0.13
GALGO+MLHD (multivariate)	BE+FS+MLHD	23	0.01
GALGO+SVM (multivariate)	FS+SVM	10	0.02
GALGO+RF (multivariate)	FS+RF	14	0.01
GALGO+KNN (multivariate)	FS+KNN	47	0.01
GALGO+NC (multivariate)	FS+NC	50	0.01

FS – Forward Selection, BE – Backward Elimination. DLDA – Diagonal Linear Discriminant Analysis, PAM – Shrunken Centroids, KNN – K-Nearest-Neighbours, SVM – Support Vector Machines, NC – Nearest Centroid, MLHD – Maximum Likelihood Discriminant Functions, RF – Random Forest.

Our results shows that GALGO tends to produce models with comparable or better classification accuracies respect to univariate variable selection strategies. The multivariate selected models generally use a smaller number of genes than univariate models. The gene identity from univariate and multivariate models appears different. Moreover, the gene identity More details are given in supplementary material in Trevino and Falciani (2006, -submitted-). This results support the use of a multivariate model selection strategy in the analysis of functional genomics data and in particularly support GALGO as a general tool.

2 Quick GALGO Tutorial

This section describes a typical application of GALGO in biomarker discovery using large scale expression profiling data. The aim of this analysis is to identify gene sets that are predictive of disease type in a panel of leukaemia patients. This tutorial will describe the main basic functionality implemented in GALGO to introduce the reader in the entire process leaving advances features in each step for further sections. The analysis pipeline implemented below is summarised in a schematic form in Figure 6.

2.1 The dataset

This tutorial uses a dataset generated by Yeoh *et al.* (2002). Briefly, samples from 327 acute lymphoblastic leukaemia (ALL) patients representing 7 different disease sub-classes have been processed to acquire expression profiling data using Affymetrix GeneChips. This tutorial summarise the results of the analysis for a five class problem. Data representing gene expression profile of five groups of patients (EMLLA, Hyp+50, MLL, T, and TEL including 27, 64, 20, 43, and 79 samples respectively) have been selected. The original dataset comprising 12,600 genes have been filtered to eliminate the most invariant genes. The standard deviation and difference between maximum and minimum expression value were calculated for each gene. The genes were ranked by these values, and if they were within the top 15% for either, were selected for further analysis. The dataset after filtering contained the expression values for 2,435 genes.

2.2 Step 1 – Setting-Up the Analysis

In the GALGO package we have included a data-frame object (ALL) that contains the normalized expression values. The object is a matrix in which rows are genes and columns are samples. The identity of the samples is defined in another object that (ALL.classes). Both objects are loaded using the function data (name object).

In R type:

```
> library(galgo)
> data(ALL)
> data(ALL.classes)
```

Of course, user data from an external text file can be loaded (see section 2.1). The wrapper function "configBB.VarSel" is used to specify the data, the parameters for the GA search, the classification method, the error estimation method, and any user-defined parameter. This function builds a BigBang

object that contains the data and the values of all parameters and will eventually stores the results of the analysis.

To set up the GA search type in R:

```
> bb.nc <- configBB.VarSel(
data=ALL,
classes=ALL.classes,
classification.method="nearcent",
chromosomeSize=5,
maxSolutions=300,
goalFitness = 0.90,
saveVariable="bb.nc",
saveFrequency=30,
saveFile="bb.nc.Rdata")
```

The code above configure a BigBang Object that will store 300 chromosomes (*maxSolutions=300*) which will contain 5 genes (*chromosomeSize=5*) that correspond to models developed using a nearest centroid classifier (*classification.method="nearcent"*) with a classification accuracy of at least 90% (*goalFitness=0.9*). The other parameters define the name of the saved object that is created (*saveVariable="bb.nc"*), the frequency of saving the results in a file (*saveFrequency=30*) and the name of the file where the results are saved (*saveFile="bb.nc.Rdata"*).

The wrapper function *configBB.VarSel* can also be used to configure additional functions. These will briefly explained in following sections. Please refer to package manual for an extensive description of the *configBB.VarSel* parameter specification. To show the parameter specification type:

```
> ?configBB.VarSel
```

2.3 Step 2 - Evolving Models/Chromosomes

Once the BigBang and Galgo objects are configured properly, we are ready to start the procedure and to collecting chromosomes associated to good predictive models of tumour class. This is achieved by calling the method "blast".

In R type:

```
> blast(bb.nc)
```

This procedure can last a long time (from minutes to hours) depending on the degree of difficulty of the classification problem, on the classification method, and on the GA search parameters. The default configuration displays the curse of BigBang and Galgo objects to the console (controlled by the verbose parameter) including the approximated remaining time.

This is an example of the text output for one GA cycle (61 generations):

```
[e] Starting: Fitness Goal=0.9, Generations=(10 : 200)
[e]     Elapsed Time     Generation     Fitness %Fit    [Next Generations]
[e]     0h 0m 0s     (m)     0     0.64103 71.23% +++++++...+.........
[e]     0h 0m 6s         20     0.87179 96.87% ....................
[e]     0h 0m 14s        40     0.87179 96.87% .....+..+.....+.+...
[e]     0h 0m 22s        60     0.92308 102.56% +
[e]     0h 0m 22s     ***     61     0.92308 102.56% FINISH: 2164 1612...
[Bb]    300     299     Sol Ok  0.92308 102.56% 61       22.16s  3722s
4054s   14 (0h 0m 14s )
```

Lines starting with "[Bb]" correspond to the current collection of the BigBang object. This line shows respectively the number of evolutions (300 in this case), the number of evolutions that have reached the goal fitness (299), the status of the last evolution (Sol Ok – the goal fitness was reached), the fitness value of the best chromosome from the last evolution (0.92408) along with it percentage relative to the goal fitness (102.56%), the number of generations it needed (61), the process time spent in last evolution (22.16 seconds), the accumulated process time spent in all evolutions (3,722 seconds), the accumulated real time (4,054 seconds, which considers the time spent by saving the object and other operative system delays), and the remaining time needed to collect the previously specified number of chromosomes (14 seconds).

Lines starting with "[e]" represent the output of the evolutionary process (the genetic algorithm search). The first line of each evolution shows the goal fitness and the constraints in generations. Successive lines show, in columns, the elapsed time, the current number of generation (by default refreshed every 20 generations) and the current best fitness along with the percentage relative to the goal fitness. The last column summarize the behaviour of next generations, "+" means that maximum fitness of the current population has increased, "-" means that it has decreased, and "." means that it has not changed. "G" appears occasionally when the fitness goal has been reach but the algorithm can not end because a constraint in the number of generations.

The default configuration would show three plots summarizing the characteristics of the population of selected chromosomes Figure 7. The topmost plot shows the number of times each gene has been present in a stored chromosome, by default the top 50 genes are coloured and the top 7 are

named. The middle plot shows the stability of the rank of the top 50 genes, which is designed to aid in the decision to stop or continue the process once the top ranked genes are stabilized. When genes have many changes in ranks, the plot show different colours; hence the rank of these genes is unstable. Commonly the top 7 "black" genes are stabilized quickly, in 100 to 300 solutions, whereas low ranked "grey" genes would require many thousands of solutions to be stabilized. The plot at the bottom is the distribution of the last generation of the GA process that have produced a solution. It is intended to show how difficult is the search problem for the current configuration of GA. If peaks are observed at either end, a configuration change is advisable (see further sections).

Once the blast method ends, you can continue with the analysis step. However, the blast process can be interrupted (by typing the ctrl-c keys in Linux or esc in windows) and the results analyzed straight away. It is recommended to break the process in the evolution stage, not in the BigBang update stage that may disrupt the object. The process can be resumed by typing the blast command again. The result of the last evolution might be lost but the accumulated results should remain be intact. Resuming the process will have the effect of restarting the Galgo object as in any cycle. The possibility to interrupt the process is very useful for initial exploratory analysis since the most updated results can be analysed and can be saved anyway using the saveObject method. Instead of interrupting the process, you can open a new R console and benefit from the use of progressive saving strategy that updates the current object called "bb.nc" into a file named "bb.nc.Rdata" once at least 30 solutions have been reached (controlled by saveVariable, saveFile, and saveFrequency parameters respectively). To do this, a previously saved object can be loaded in GALGO using the loadObject method in a new R console window:

```
> library(galgo)
#change directory to yours
> loadObject("bb.nc.Rdata")
```

Once the file is loaded, the loadObject method displays a summary of the loaded variables and their classes and you can proceed to the analysis step.
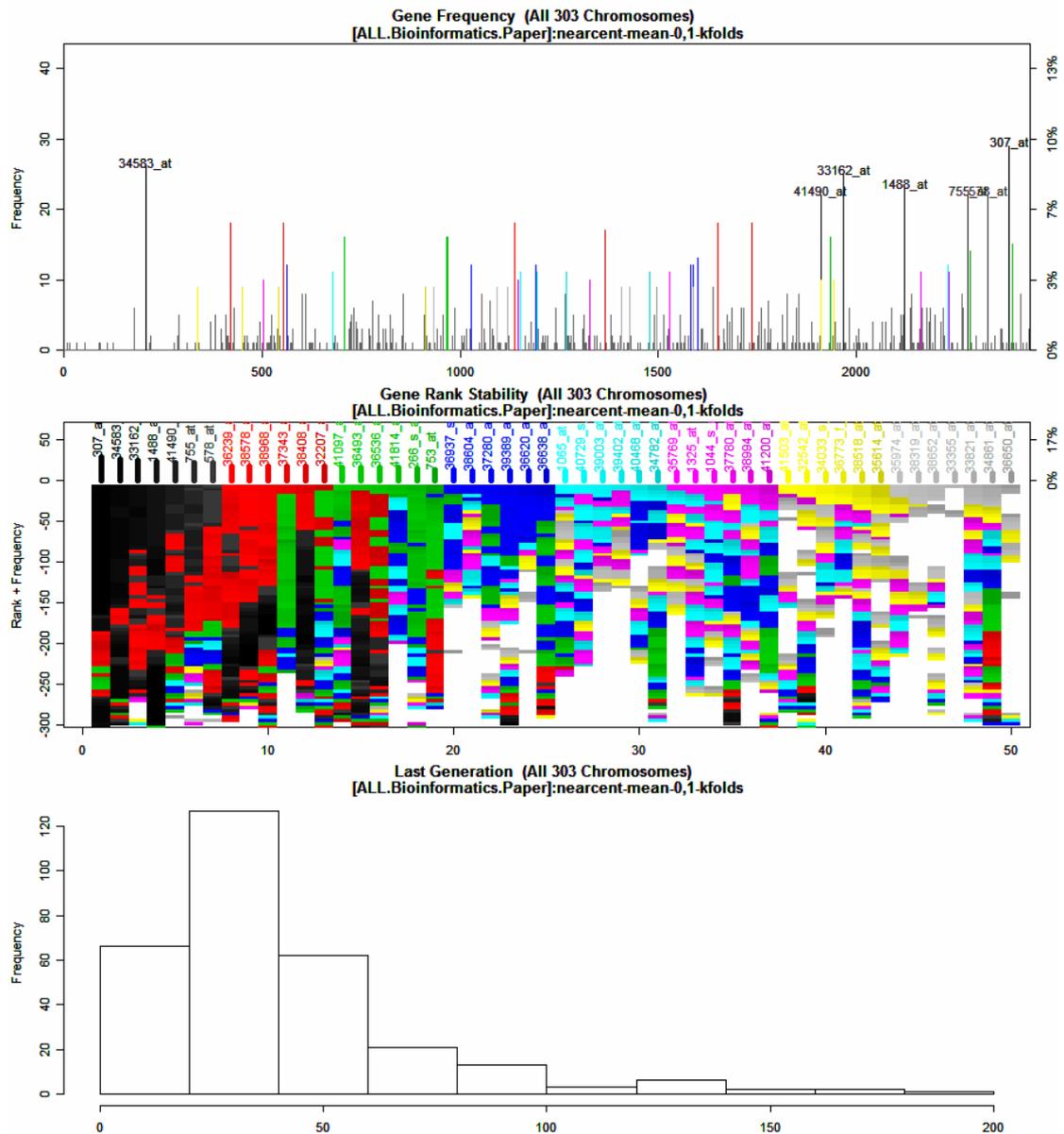
Figure 7 Default monitoring of accumulated chromosomes in the BigBang object.

GALGO also have the functionality to summarise the population of chromosomes within each generation. The code below shows the modifications to the definition of the BigBang Object that are required to activate this function (marked in red).

```
> x11()
> x11()
> bb.nc <- configBB.VarSel(
data=ALL,
classes=ALL.classes,
classification.method="nearcent",
chromosomeSize=5,
maxSolutions=300,
goalFitness = 0.90,
saveVariable="bb.nc",
```

```
saveFrequency=30,
saveFile="bb.nc.Rdata",
callBackFuncGALGO=plot,
callBackFuncBB=function(...){dev.set(2);plot(...);dev.set
(3); }
)
```

The topmost plot in Figure 8 shows the current values of the genes in chromosomes in order to show the explorative process. The middle plot shows the evolution of the fitness relative to the goal in the course of generations. The plot at the bottom shows the history of the maximum chromosome.
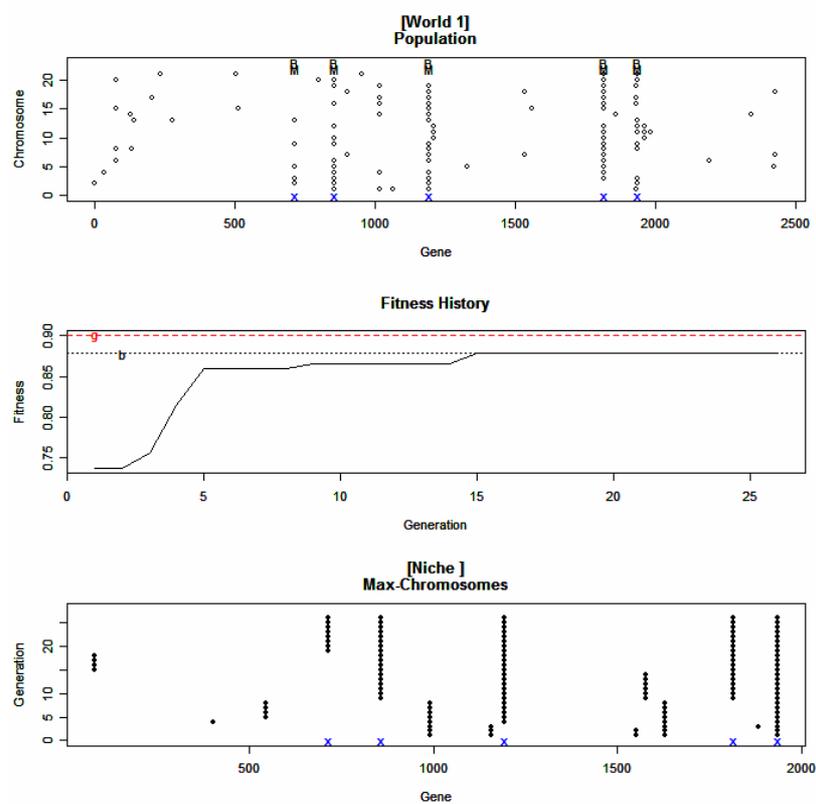
**Figure 8 Real-time monitoring of the Genetic Algorithm search. The horizontal axis of the top and bottom plots display unranked gene indexes. The vertical axis of the top panel is displaying the chromosome index whereas the vertical axis of the bottom panel is displaying the generation number. In the middle plot the horizontal axis is displaying the generation whereas the vertical axis is displaying the fitness value.**

2.4 Step 3 - Analysis and Refinement of Populations Chromosome

2.4.1 Are we getting solutions?

The first question we have to answer is whether we are actually getting acceptable solutions. By default, configBB.VarSel configures the BigBang object to save all chromosomes even if they didn't reach the goalFitness value. The reason is that we need to assess the success of the configured GA search under all searches, not only in those that reach solutions. We can analyze the success of the configured GA search by looking at the evolution of the fitness value across generations, using the code below.

```
> plot(bb.nc, type="fitness")
```

Figure 9 shows that in average, we are reaching a solution in generation 40, which is very sensible. The lines show the average fitness for all chromosomes and for those that have not reached a goal respectively. These lines intend to delimit an empirical "confidence interval" for the fitness across generations. The characteristic plateau effect could be useful to decide if the search is not working to reach our goal, which is marked with dotted line (see section 8.3 if you cannot reach solutions).
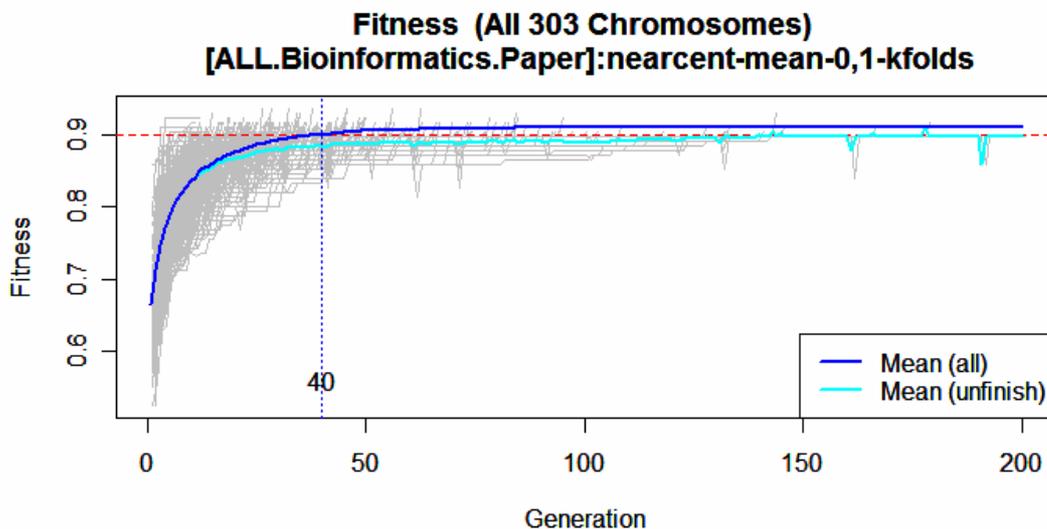

Figure 9 Evolution of the maximum fitness across generations in 303 independent searches.

It is possible to separate the evolutions that have reached the goal using the following code.

```
> par(mfrow=c(2,1))
> plot(bb.nc, type="fitness", filter="solutions")
> plot(bb.nc, type="fitness", filter="nosolutions")
```

The "filter" parameter can be used almost in any function and in any plot type.

2.4.2   What is the overall accuracy of the population of selected models?

Once the chromosomes have been selected we need to asses the classification accuracy of the corresponding models using one of the three Strategies that we describe in **BOX 1**. The default configuration will estimate the accuracy of the models using Strategy 3 as described in **BOX 1**.

Use the following command to plot the overall accuracy.

```
> plot(bb.nc, type="confusion")
```

The output of this function is shown in Figure 11. The horizontal axis represents the individual samples grouped according to the disease class whereas the vertical axis represents the predicted classes. The barcharts represent the percentage of models that classify each sample in a given class. For example, samples in second column (marked in red) belong to the HYP+50 class. These are, on average, correctly classified 85.6% of the times. However, on average, they are "wrongly" classified 2.5% of the times as EMLLA, 5.4% of the times as MLL, 1.5% as T, and 5% as TEL. The plot also report the value of sensitivity and specificity of the prediction. These are measures of the overall prediction per class. The sensitivity of the prediction for a given class k is defined as the proportion of samples in k that are correctly classified. The specificity for a given class k is defined as the number of true negatives divided by the sum of true negatives and true positives.

To obtain the confusion matrix, specificity, and sensitivity measures in a numeric format use the following code.

```
> cpm <- classPredictionMatrix(bb.nc)
> cm <- confusionMatrix(bb.nc, cpm)
> sec <- sensitivityClass(bb.nc, cm)
> spc <- specificityClass(bb.nc, cm)
```

BOX 1: Error estimation Strategies in GALGO

There are several methods to estimate Classification accuracy. These are all based on the fundamental principle that a correct estimate of accuracy must be performed on a set of samples that has not been used to develop the model itself.

Classical approaches involve splitting data in training and test sets. The training set is used to estimate the parameters of the model whereas the test set is left aside and it is used to asses the accuracy of the model itself. This approach is considered the most appropriate when a large number of samples is available. However, when the number of samples is relatively small, as it is the case of a typical microarray experiment, the test set could be too small to estimate the classification accuracy with acceptable precision.

In order to estimate the accuracy with small datasets it is possible to use a different statistical technique called *cross-validation*. The dataset is split in *k* different training and test sets. The classification accuracy is then defined as the average of the classification accuracies calculated, by default, on the test sets for each of the *k* splits. GALGO uses a technique called bootstrapping (Efron *et al.*, 1993) to generate the splits.

Within GALGO we can use three main strategies for estimating Classification accuracy. In the first strategy a simple cross-validation or resubstitution error strategy is used to compute the value of the fitness function that guide chromosome selection in the GA procedure. The classification accuracy of the selected chromosome is defined as the fitness value (Figure 10A). The second strategy (Figure 10B) is a classic Training and Test procedure where the accuracy is estimated on the test data. In the GA process, the value of the fitness function is estimated by cross validation on the training data. Other approaches, such as .632 bootstrap (Efron *et al.*, 1993), combine training and test accuracies, which can be specified as error weights through the parameter *classification.test.error = c(.368, .632)* for training and test respectively. The third strategy is to select the Chromosomes as in the second strategy and to compute the classification accuracy of the selected chromosomes as the average of the classification accuracy estimated on *k* data splits as exemplified in Figure 10C.

GALGO defines the initial split (common to both strategies) as Split 1.

A

(no splits)

DATA

Training And Test

K-fold-cross-validation
resubstitution
loocv

Chromosomes

train=1
test=0

B

Split 1

DATA

2/3 | 1/3

Training | Test

K-fold-cross-validation
resubstitution
loocv

Chromosomes

train=2/3
test=1/3

C

Split 2 – Split k:

DATA

2/3 | 1/3

Training | Test

K-fold-cross-validation
resubstitution
loocv

Chromosomes

train=rep(2/3,k)
test=rep(1/3,k)
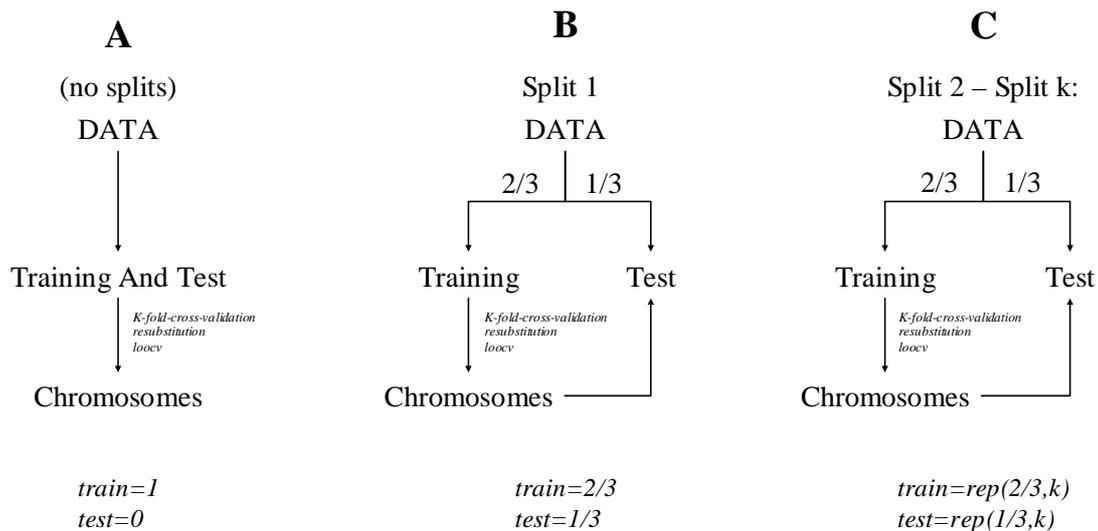
Figure 10 Schematic Representation of the Estimation of Classification Accuracy. (A) Strategy 1, using all data as training and test. (B) Strategy 2, classical training and test. (C) Strategy 3, *k* repetitions of the strategy 2. The respective values of the parameters, *train* and *test*, needed to perform each strategy is shown at the bottom of the schema.

cpm is a matrix with the number of times every sample as been predicted as any other class. For instance, let analyze the first rows of cpm.

```
> cpm[1:3,]
          EMLLA HYP+50  MLL    T TEL (NA)
E2A.PBX1.C1 10837   1418 1062 1473 663    0
E2A.PBX1.C2 13654    101  767  113 212    0
E2A.PBX1.C3 13729    262 1225  218  19    0
```

The output above shows that E2A.PBX1.C2 sample has been predicted 14847 times (the row sum), in which 13654 times (92%) as been predicted as EMLLA, 101 times (0.6%) as HYP+50 and so on. The number of predictions depends on how we estimate the error in terms of training and test sets, and the number of chromosomes (type ?classPredictionMatrix.BigBang). By default, the prediction is made on test sets only for each chromosome (303 in the plot shown here). Initially, configBB.VarSel function generated 150 random test sets and each test set was made using 1/3 of the samples. Thus, in average, a sample would be predicted 303 * 150/3 times, that is 15150 times, which is comparable to 14847 for the second sample. In certain circumstances, some classification methods cannot make a prediction based on the data, "(NA)" column summarise those cases (for nearest centroid method, it will be 0 always).
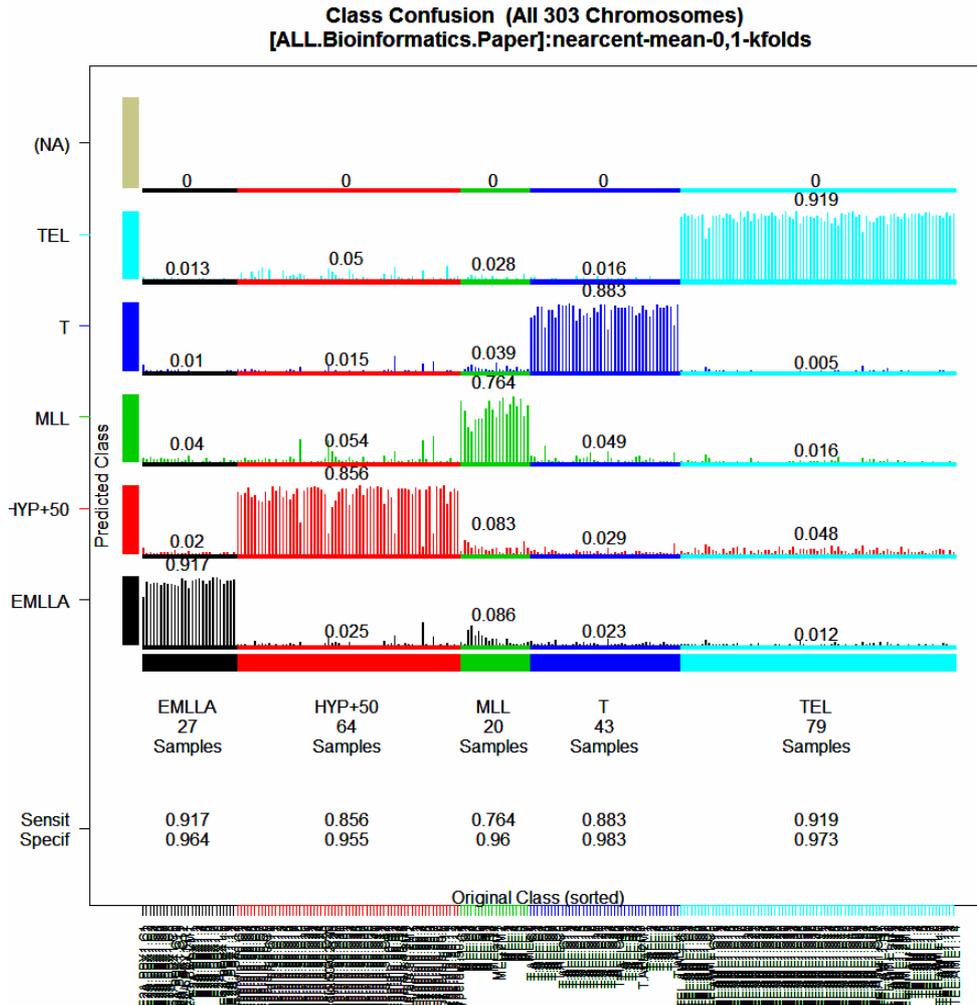
Figure 11 Overall classification accuracy.

To evaluate the error in the first training set (as they were evolved), we can use the following changes in parameters.

```
> plot(bb.nc, type="confusion", set=c(1,0), splits=1,
filter="solutions")
```

set parameter specify that the error estimation should be computed in the training set only. *splits* parameter limit the estimation to one partition, the original used to evolve the chromosomes. *filter* specify that only chromosomes that reach the goal fitness will be evaluated. In this plot (not shown) some samples do not show it respective "bar", which indicate that those samples were never predicted. This is because we limit the evaluation to the train set in the split #1, which should contain 155 samples approximately (66%).

To evaluate a single chromosome or any other model in the same circumstances use the following code.

```
> plot(bb.nc, type="confusion", set=c(0,1), splits=1,
chromosomes=list(bb.nc$bestChromosomes[[1]]))
```

In this case, the bars do not represent an average prediction because each test sample were predicted once (1 model in 1 split only).

2.4.3   Is the rank of the genes stable?

Stochastic searches (such as GA) are very efficient methods to identify solutions to an optimization problem (e.g. classification). However they are exploring only a small portion of the total model space. The starting point of any GA search is a random population. Different searches therefore are likely to provide different solutions. In order to extensively cover the space of models that can be explored it is necessary to collect a large number of chromosomes. GALGO offers a diagnostic tool to determine when the GA searches reach some degree of convergence. Our approach is based on the analysis of the frequency that each gene appears in the chromosome population. As chromosomes are selected the frequency of each gene in the population will change until no new solutions are found. Therefore monitoring the stability of gene ranks (based on their frequency) offers the possibility to visualize model convergence.

To produce the rank stability plot type:

```
> plot(bb.nc, type="generankstability")
```

By default, the most frequent 50 genes are shown in 8 different colours with about 6 or 7 genes per colour (Figure 5). Horizontal axis in Figure 5 shows the genes ordered by rank. Vertical axis shows the gene frequency (in the top part of the y axis) and the colour coded rank of each gene in previous evolutions. Consequently, for a given gene, changes in ranks are marked by different colours (below the frequency). Figure 5 shows that the first 7 black genes have been stable at least during the last 50 solutions whereas some red genes have recently swap from green. Thus, red and green genes are not yet stable; this is because 303 chromosomes are not enough to stabilize these genes. Probably, 1000 chromosomes would generate more stable results, however, the more chromosomes the better. For comparison, Figure 6 shows the result for the same run used here but using 1000 chromosomes, which exhibit more stability in ranks. Another property is that top genes are being stabilized in order; first black genes, then red, green and so on. For longer runs comparisons, see further sections.
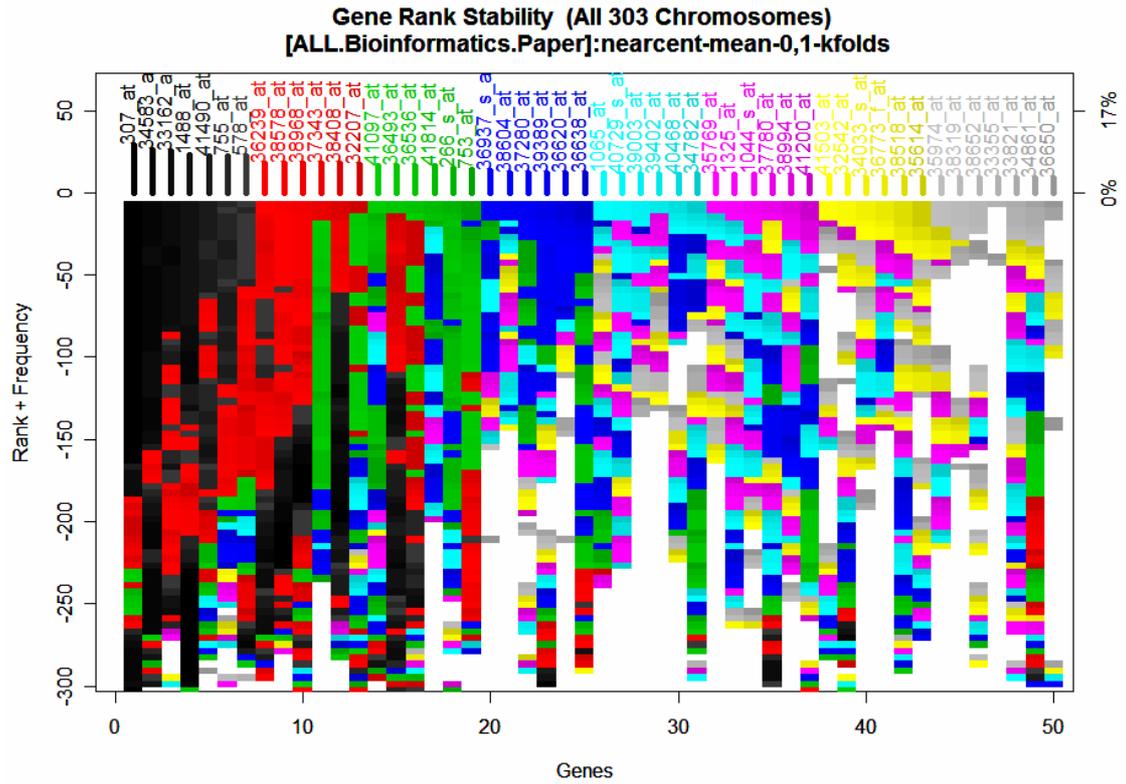
**Figure 12 Gene Ranks across past evolutions.**
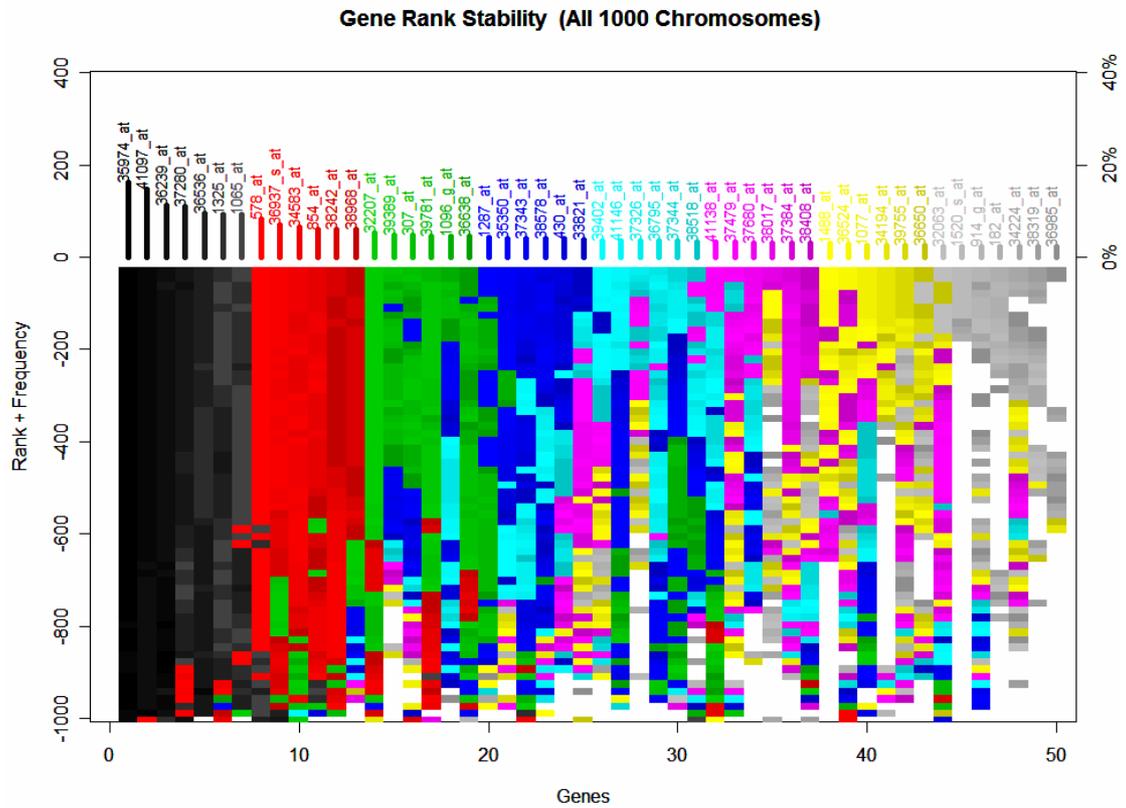
Figure 13 Rank Stability in 1000 chromosomes.

2.4.4   Are all genes included in a chromosome contributing to the model accuracy?

The chromosome size is fixed by an initial parameter in GALGO. This implies that some of the genes selected in the chromosome could not be contributing to the classification accuracy of the correspondent model. GALGO offers the possibility to identify these genes and remove them from the chromosomes. This can be done after the selection is completed or within the selection process itself. In order to perform this task we have implemented a backward selection procedure. The methodology works as follows. A given gene is removed from the chromosome. The classification accuracy of the resulting shorter chromosome is then computed. If this is not reduced, another elimination cycle is performed. If the Classification accuracy is reduced the gene is left in the chromosome and another elimination cycle is performed until all genes have been tested.

In order to perform this procedure type:

```
> rchr <- lapply(bb.nc$bestChromosomes[1:300],
robustGeneBackwardElimination, bb.nc, result="shortest")
```

The distribution of the size of the refined chromosome population can be plotted using the following function.

```
> barplot(table(unlist(lapply(rchr,length))),
main="Length of Shortened Chromosomes")
```
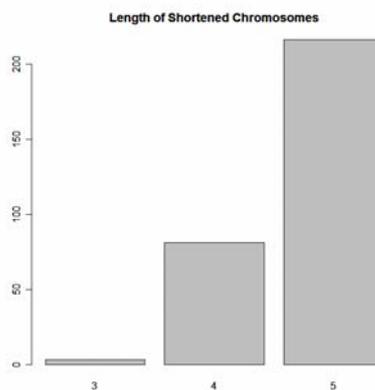

Figure 14 Refinement of the chromosomes.

Figure 14 shows that a large proportion of the chromosomes require all five genes to accurately classify the samples. Considering that the problem we are trying to solve here is a five-class problem (multi-class), the fact that in this example the majority of the models actually need five genes is not particularly

surprising. However, it is common to build models with more genes than classes; indeed the majority of the datasets actually contain only two classes (e.g. treated-untreated, cancer-normal, wild-mutant, etc). Therefore we encourage the users to perform this analysis regularly.

2.5   Step 4 - Developing Representative Models

The GA procedure provides us with a large collection of chromosomes. Although these are all good solutions of the problem, it is not clear which one should be chosen for developing a classifier, for example, of clinical importance or for biological interpretation. For this reason there is a need to develop a single model that is, to some extent, representative of the population. The simpler strategy to follow is to use the frequency of genes in the population of chromosomes as criteria for inclusion in a forward selection strategy. The model of choice will be the one with the highest classification accuracy and the lower number of genes. However GALGO also stores alternative models with similar accuracy and larger number of genes. This strategy ensures that the most represented genes in the population of chromosomes are included in a single summary model.

This procedure should be applied to the population of chromosomes generated by initial GA search. However, it can also be applied to the population of chromosomes that is the result of backward selection procedure explains in the previous paragraph.

The forward selection model can be generated by typing:

```
> fsm <- forwardSelectionModels(bb.nc)
> fsm$models
> ?forwardSelectionModels.BigBang # Help System
```
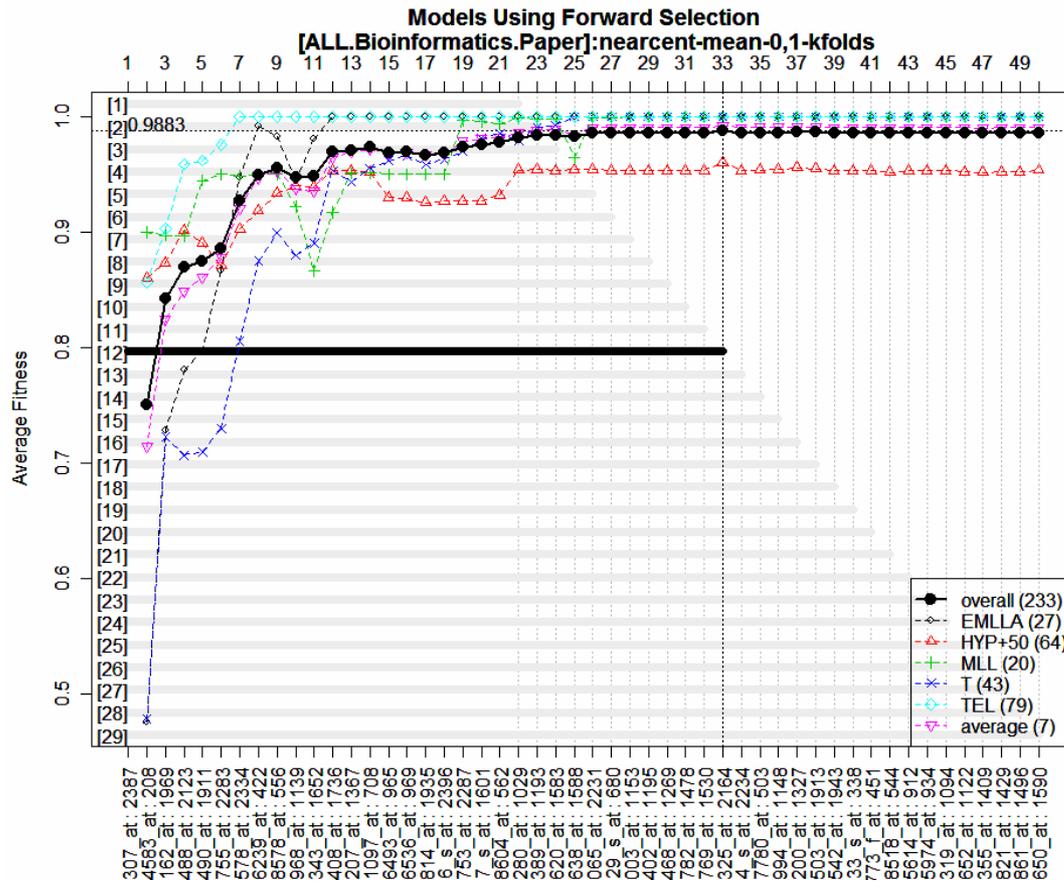
Figure 15 Forward selection using the most frequent genes. Horizontal axis represents the genes ordered by their rank. Vertical axis shows the classification accuracy. Solid line represents the overall accuracy (misclassified samples divided by the total number of samples). Coloured dashed lines represent the accuracy per class. 1 model resulted from the selection whose fitness value is maximum (black thick line), but 29 models were finally reported because they were very similar in absolute value.

Figure 15 shows the results from forward selection procedure. The selection is done evaluating the test error using the fitness function in all test sets. The output is a list of values including the models whose fitness is higher than 99% of the maximum (or above a specified value using "minFitness" parameter). *fsm* object contains the **best** models (29 in this case). Model labelled as 12, containing the most 33 frequent genes, was the best model in terms of accuracy. The other 28 models included in *fsm* are 99% as close to the best model. Any resulted models can be viewed in heat maps, PCA space, or profiles. To visualize the best model in a heatmap plot use the following code.

```
> heatmapModels(bb.nc, fsm, subset=12)
```

Details for visualization of models (or chromosomes) are given in sections 2.6 and 5.6.

The classification accuracy can be plotted extracting the information for any of these specific models, as in the example below (plot not shown).

```
> plot(bb.nc, type="confusion",
chromosomes=list(fsm$models[[1]]))
> cpm.1 <- classPredictionMatrix(bb.nc,
chromosomes=list(fsm$models[[1]]))
> cm.1 <- confusionMatrix(bb.nc, cpm.1)
> mean(sensitivityClass(bb.nc, cm.1))
[1] 0.9863334
> mean(specificityClass(bb.nc, cm.1))
[1] 0.9965833
```

From the mean values of sensitivity and specificity we can conclude that the selected model is, by far, more accurate than any original evolved chromosome.

2.6 *Visualizing Models and Chromosomes*

Gene signatures associated within individual chromosomes or in a representative model (derived by forward selection) can be visualised in GALGO using a number of graphical functions. In this section, we will demonstrate the use of heat maps and PCA. For, the typical heat map format, use the following commands.

```
> heatmapModels(bb.nc, fsm, subset=1) # forward
> heatmapModels(bb.nc, bb.nc$bestChromosomes[1])
```

The results are shown in Figure 13[*].

In order to visualise the relation of samples using the genes selected in a chromosome or in a representative model we can also use principal component analysis representation. In order to do this, type the following command (Figure 17).

```
> pcaModels(bb.nc, fsm, subset=1)
> pcaModels(bb.nc, bb.nc$bestChromosomes[1])
```

---

[*] Remember that the hierarchical clustering of samples given in the heat map is the product of an unsupervised algorithm, which may differ from the classification method of our choice. Therefore, the relative sample order in the heat map, the original class, and the predicted class by the model may all be different. Nevertheless, many of the times, the hierarchical clustering gives a good overview.

By default, only the first four components are shown, which can be changed specifying the *npc* parameter.

Other useful way to show a model is using the profiles of samples within a class as shown in Figure 18, which is the result from the code.

```
> plot(bb.nc, fsm$models[[1]], type="sampleprofiles")
```
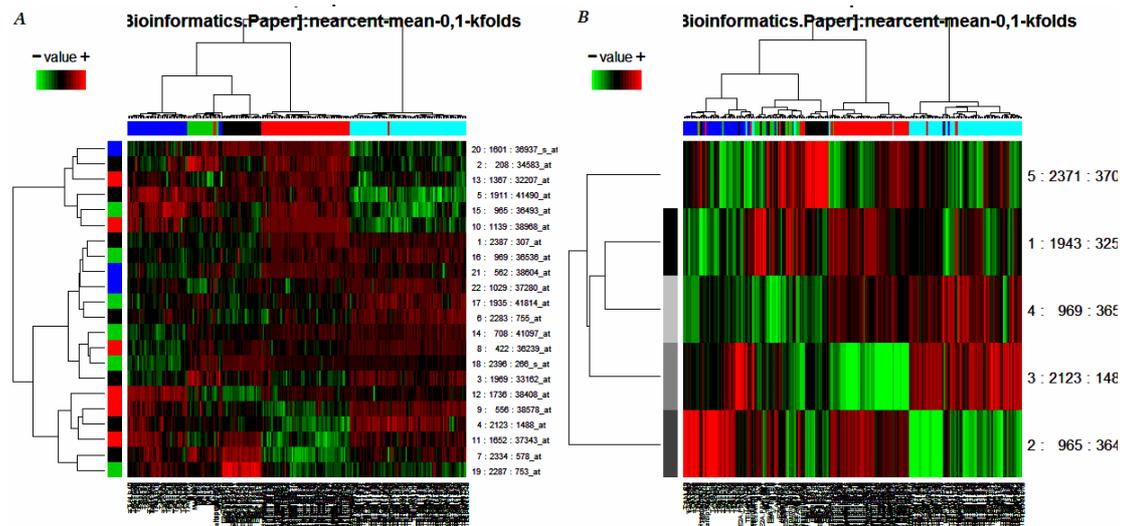

Figure 16 Heatmaps. From a model resulted from forward selection (A) and an original evolved chromosome (B).
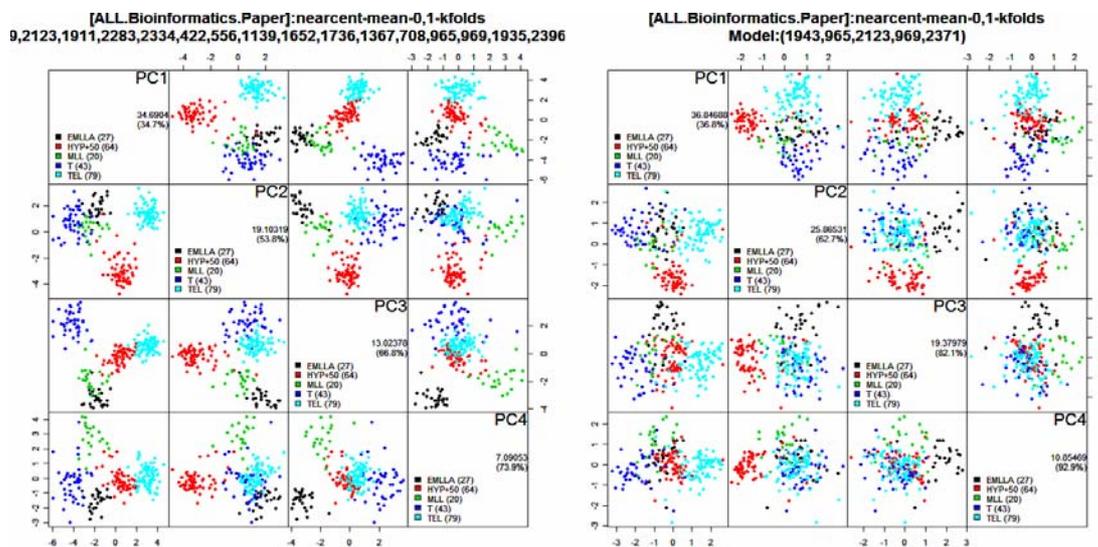

Figure 17 Depiction of a model (left) and a chromosome (right) in PCA space.
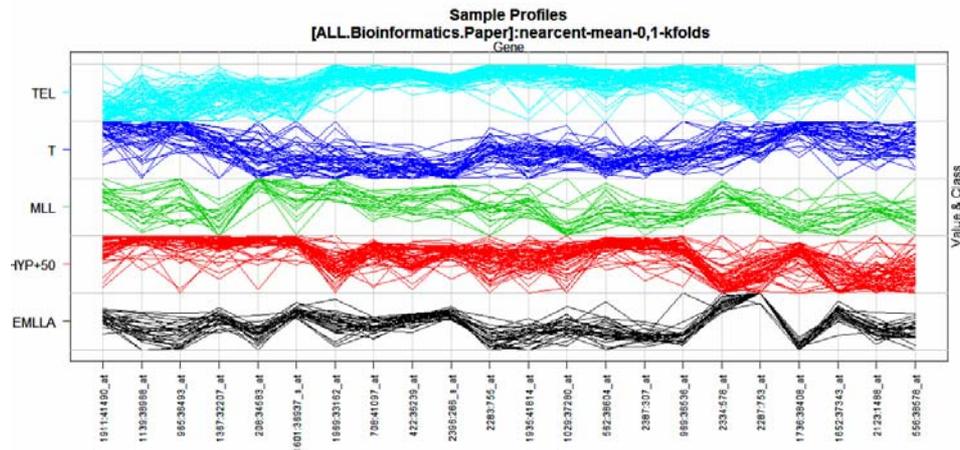
Figure 18 Sample profiles per class.

2.7 *Predicting Class Membership of Unknown Samples*

An important characteristic of any model is their ability to make predictions. Models designed using GALGO can be evaluated with a complete unknown or blind dataset (see also section 5.7). The following code exemplify how make predictions in a new "dummy" dataset for all chromosomes collected in the BigBang object.

```
> data(ALL)
# dummy data: the first 15 samples from original ALL data
# which all must be from EMLLA class
> dummy <- ALL[,1:15]
> ?predict.BigBang
> cpm <- predict(bb.nc, newdata=dummy,
func=classPredictionMatrix, splits=1:10)
> cpm
> plot(bb.nc, cpm, type="confusion")
```

In the above code, *dummy* was temporally appended to the original data. Then *classPredictionMatrix* was run for all chromosomes. *splits* is a parameter used in *classPredictionMatrix* (which was used to illustrate the use of user-parameters for any function specified in *func*). The result of the plot is shown in Figure 19 where the new data was labelled as "UNKNOWN". The black bars in these samples indicate that they were predicted as EMLLA (as expected).

To predict new data using an individual model, we may use the *classPredictionMatrix* method using the *chromosomes* parameter (see ?classPredictionMatrix.BigBang), such as in the following code.

```
> cpm <- predict(bb.nc, newdata=dummy,
func=classPredictionMatrix, chromosomes=fsm$models[1])
> cpm
> plot(bb.nc, cpm, type="confusion")
```
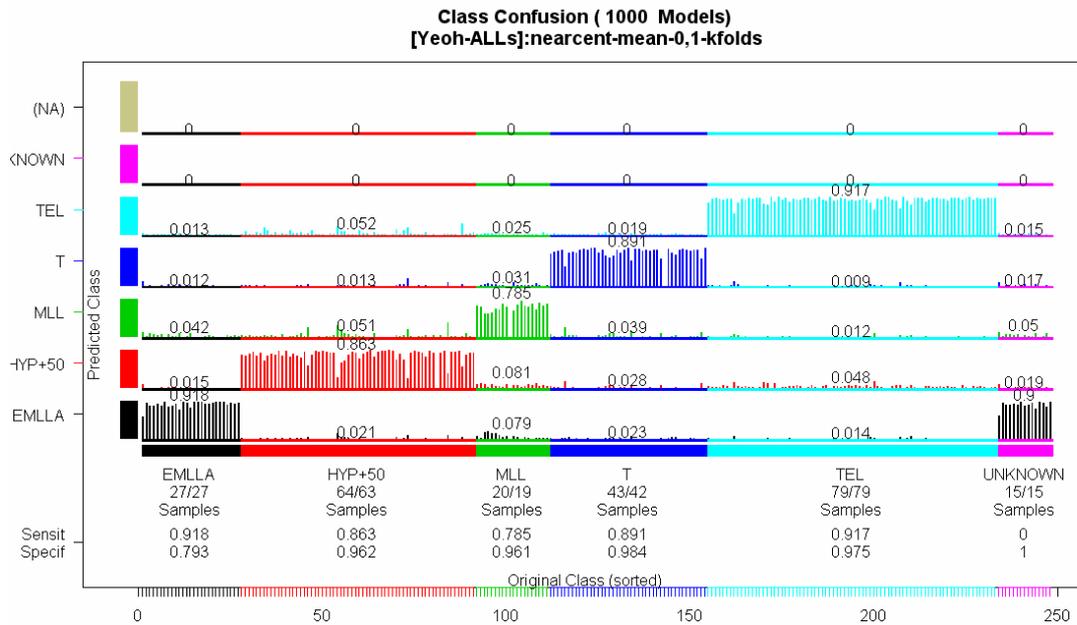

Figure 19 prediction for unknown samples (the last 15 samples in the right).

2.8 Summary

The *configBB.VarSel* configures the necessary objects and specifies the right parameters to make the entire process to work in different contexts and testing strategies with the classification method of your choice. In addition, the implementation of new classification methods is simplified providing your specific fitness function in the *classification.userFitnessFunc* parameter (type ?configBB.VarSel).

We have seen in this section how to build multivariate statistical models for a classification problem using GALGO. So far, we have setup and performed a basic analysis with the dataset included. In what follows is a more advanced analysis explaining many of the available options in each step that can be customized for particular data, classification methods, GA searches, user defined fitness functions, error estimation, process parallelization, GA parameters, and troubleshooting. In these sections it will be assumed that the "quick tutorial" has been executed and that the reader is familiarised with the concepts expressed there.

3 Step 1 - Setting-up the Analysis

Once the reader is get used with a common GALGO run, we can introduce more advanced customizations. Many of the options shown here are derived from the help description in *configBB.VarSel*, *configBB.VarSelMisc*, and BigBang. Examples for accessing the R help system are as following.

```
> library(galgo)
> ?configBB.VarSel
> ?configBB.VarSelMisc
> ?BigBang
> ?plot.BigBang
> ?confusionMatrix.BigBang
```

3.1 Data Source

The most common data source is a text file with tab delimited file (easily processed in excel or any other spreadsheet program). The expected file format is genes in rows and samples in columns. The first column must be gene names, identifier, accession number, or anything to distinguish uniquely the genes. The first row must contain the sample names, again unique values. The second row is the class description for each sample, but it can be optionally provided in a separated parameter. A common data file looks like the following.

```
          Sample1    Sample2    Sample3    Sample4    Sample5
class     classA     classA     classB     classC     classC
36237_at  10.2266    10.1613    12.0972    11.8012    10.9888
36238_at  10.2464    9.59952    9.00217    10.3131    11.2178
36239_at  12.4709    13.1544    13.3683    13.3076    12.7597
...
```

To load this file, instead of the *data* parameter that is used when the data is provided in an R matrix or data frame object, we use the *file* parameter as shown below.

```
bb.nc <- configBB.VarSel(file="myfile.txt", ...)
```

configBB methods assumes that data has been normalized previously. GALGO do not provide normalization methods to correct systematic errors, therefore the user must normalize the data before the analysis. However, some classification methods require that the data is standarized (*mean=0, variance=1*), so careful attention must be paid to the *scale* parameter in order to force *configBB* methods to standardize or not to standardize your data. By

default, *scale* is performed when the classification method is "knn", "nearcent", "mlhd", or "svm".

3.2 Classification Method

Currently, GALGO provides the coupling with 6 classification methods that cover a broad spectrum of statistical and artificial intelligence techniques. These are maximum likelihood discriminant functions (MLHD), k-nearest-neighbours (KNN), nearest centroid (NEARCENT), classification trees (RPART), support vector machines (SVM) and neural networks (NNET). These methods can be computationally intensive; therefore, in order to improve performance, we have implemented MLHD, KNN and NEARCENT in the programming language C. We recommend their usage over the others if time is critical. All methods except MLHD require specific parameters. In the tutorial (section 2), we have use the nearest centroid method in order to compare the results with a similar univariate method [application notes paper in bioinformatics].

3.2.1 MLHD

Discriminant analysis is a powerful technique designed to distinguish between samples groups initially proposed by Fisher (Lattin *et al.* 2003). This technique is equivalent to MANOVA turned around (Tabachnick *et al.* 2001). Discriminant analysis considers the covariance and means of multiple variables. The original proposed discriminant analysis is a linear combination of the original variables that maximize the separation (linear discriminant analysis).

MLHD refer to Maximum LikeliHooD, where Bayes' rule is used as the discriminant function designating a sample to the class with maximum conditional probability. MLHD in the context of microarray classification was first used by Ooi *et al.* (Ooi *et al.* 2003). The discriminant function relies on the means for every gene in the GA chromosome for all classes and the pooled covariance matrix for all genes in GA chromosome for all classes. The mathematical relations are:

$$f_q(e) = \mu_q^T \Sigma^{-1} e - \frac{\Sigma^{-1} \mu_q^T}{2}$$

$$\Sigma = \frac{1}{M_t - Q} \sum_{q=1}^{Q} \Sigma q$$

$$c = q \,|\, \max(f_k(e)) = f_q(e)$$

where μ_q is the mean vector in class q, Σ is the pooled covariance matrix, Σ_q is the covariance matrix for class q, M_t is the number of training samples and Q

is the number of classes. A unknown sample class *c* is designated as class *q* for that $f_q(e)$ which is maximum.

MLHD works better with standardized data. Therefore the default for *scale* parameter in *configBB.VarSel* is activated. We made a GALGO run with non-standardized data and it still works. For *configBB.VarSel*, the parameter needed is *classification.method="mlhd"*. GALGO uses this classification method through the included *mlhd.C.predict* and *mlhd.R.predict* methods.

3.2.2 K-Nearest-Neighbour

For a given sample the nearest neighbour is the sample that is the closest. Hence, a measure of distance is needed to determine how close two samples are. Because of its generality, euclidean distance is preferred as distance measure which is:

$$d_{ab} = \sqrt{\sum_{i=1}^{G} (x_{ia} - x_{ib})^2}$$

where G is the number of genes in the chromosome and a and b are samples. For an unknown sample, the distance with all known samples is computed and sorted. The smallest distance is the first nearest neighbour; the second smaller is the second nearest neighbour and so on until consider the *k* closest samples to the one being classified. The unknown sample is designated in the same class than the majority of all *k* nearest samples. When there is a draw, the sample is considered unclassified. If only two classes are used an odd value for *k* is recommended to avoid a draw. A more stringent condition would be that all *k* samples must pertain to same class, this force GA engine search for better genes. It is important to experiment with both schemes because the desired scenario is when all nearest neighbour are exactly the same class, however sometimes is very difficult achieve this scenario.

The parameters needed for *configBB.VarSel* are *classification.method="knn"*, *knn.k* which is the number of neighbours to consider, *knn.l* which is the minimum number of neighbours of the same class *C* needed to predict the sample as class *C*, and *knn.distance*, which is the distance method used to look for neighbours. The default values are 3, 1, and *"euclidean"* for *knn.k*, *knn.l*, and *knn.distance* respectively. Several distance methods are available (see *dist* and *configBB.VarSel* methods). KNN method is non-parametric, thus it does not require the data follows a normal distribution. However, it is recommended to standardize the data before the analysis, therefore *scale* parameter is by default activated. GALGO uses this classification method through the included *knn.C.predict* and *knn.R.predict* methods.

3.2.3 Nearest Centroid

For a given set of samples, the centroid is defined as the mean or median value. The nearest centroid for an unknown sample is the centroid whose euclidean distance is minimum.

The parameters needed for *configBB.VarSel* is *classification.method="nearcent"* and *nearcent.method*, which specify *"mean"* or *"median"* centroids. Nearest centroid method is non-parametric, thus it does not require the data follows a normal distribution. However, it is recommended to standardize the data before the analysis, therefore *scale* parameter is by default activated. GALGO uses this classification method through the included *nearcent.C.predict* and *nearcent.R.predict* methods.

3.2.4 Classification Trees

When expression levels of gene *i* in all samples from class *A* are different from samples in class *B*, the expression to classify a new sample could be viewed as if gene *i* expression level is greater than *x* then new sample would be of class *A* otherwise new sample would be of class *B.* This approach is commonly used in clinical diagnostics for proteins or metabolites. Sometimes a high proportion of samples could be classified in this very simplistic approach, however if the classes are not different enough, this approach fails in suitable classification problems. A more generalized approach was derived using several expression and variables in cascade. For instance, if gene *i* expression level > *5* then class *A* else if gene *k* expression level < *4* then class *B* else class *A*. In statistics this is called Tree Classifiers because it creates a tree structure where the first node is the first expression and the branches are the results of the expressions which in turn could be a new expression or a node called terminal. The terminal node has no variable relation; instead the final decision for the class is related to the node. This classifier is commonly used because the interpretation involving logical relation between variables, values, and classes is very simple.

GALGO use the package *rpart* to build the classification tree through the included function *rpart.R.predict*. The parameter needed for *configBB.VarSel* is *classification.method="rpart"*.

3.2.5 Neural Networks

In the brain, every neuron has axons and dendrites ends that are used to communicate neurons. The basic function of neurons cells is produce a signal response in all axons ends based on whether the stimulus signals in dendrite

ends is greater than a potential action. Inspired on this knowledge, artificial neurons have been defined as a mathematical function that produces an output based on a weighted sum of inputs if the sum is greater than a threshold (Mcculloch *et al.* 1990):

$$u(t) = \sum_{j=1}^{N} w_j(t) x_j(t)$$

$$y(t) = f(u_{rest} + u(t))$$

where u_{rest} is the resting potential, $x_j(t)$ is the j input stimulus for sample t, $w(t)_j$ is the connection strength of stimulus j, $u(t)$ is the total stimulus, f is the function that transform total stimulus to output $y(t)$. To train the neuron the inputs are presented at $x_j(t)$ and the output $y(t)$ is computed and compared with the desired output thru relation:

$$\Delta w_i(t) = (D - y(t)) x_i(t)$$

Then in order to "learn" the weight are updated as:

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

Where $w_i(t+1)$ represent the new learned weight. The initial weights were initialized to random values. When several artificial neurons are interconnected it is called a neural network and different connection configurations require slightly different or even specific learning algorithms. The most common configurations are fixed in layers, where neurons from one layer are connected with all neurons in previous and next layer but not connected with neurons in same layer.

GALGO uses the package *nnet* to build neural networks as classifiers through the function *nnet.R.predict*. The parameters needed for *configBB.VarSel* is *classification.method="nnet"*, *nnet.size*, which is the number of units in the hidden layer, *nnet.decay*, which is used for the weight training, *nnet.skip*, which are used to add skip-layer connections from input to output, *nnet.rang*, which are the range of the initial random weights (for details see *nnet* and *nnet.R.predict* methods).

3.2.6 Support Vector Machines

Suppose that we plot every sample in a convenient plane distinguishing sample class. In support vector machines (SVM) the convenient plane is called kernel function and its purpose is to transform the data into a higher dimensional space that allows a better and easier separation (Moore 2001; Smola 2000). In PCA the kernel function is a linear transformation whilst in SVM the kernel function can be customized. For classification purposes, the best line that separate classes is that whose distance between the line and the closest samples within classes are maximum (because it allows the highest margin between those points). This margins concept has lately been

recognized as a unifying principle for analyzing many different problem of learning (Smola 2000). Hence SVM is now part of discipline called margin classifiers.

GALGO use the package *e1071* to build the classification tree through the included function *svm.R.predict*. *svm* require several parameter, we have used default values for many of them and leave available the more usually needed. The parameters needed for *configBB.VarSel* are *classification.method="svm"*, svm.kernel, which specify the kernel transformation ("radial", by default), *svm.type*, *svm.nu*, *svm.degree*, and *svm.cost* (see *svm*, *svm.R.predict*, and *svm.C.predict* methods for further details).

3.2.7   Classifiers Comparison

To investigate the effect of different methods in the same data under the same conditions, we made a comparison collecting 1000 solutions using knn, nearest centroid, mlhd, and svm. Figure 20 shows that, in overall there is a good overlap between the top genes, however, each method have their own preferred genes (most frequent genes changes slightly in ranks). However, Figure 21 shows that the fitness evolutions are very different. Note that in the case of SVM, a solution is found in average in 13 generations, where in knn it last 89. In general, SVM is a very powerful method; unfortunately, it is very slow.
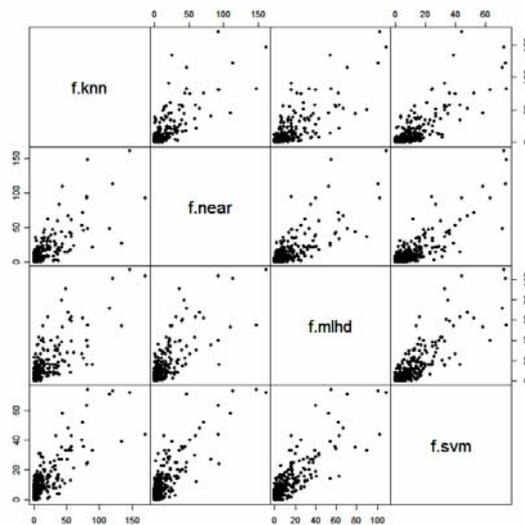

Figure 20 Gene Frequency Comparisons of Different Classifiers.

3.2.8   User-Specific Classifier

One of benefits of GALGO in R is that it is very easy to create new methods based on the vast methods available. We have designed GALGO in such a way to be flexible for the user to specify a third party classification method. This can be attained by using the parameters *classification.method="user"*, and

classification.userFitnessFunc. However, the function specified there must follow certain rules. It must receive five parameters, *chr* which receive the genes, *parent* which receive the BigBang object, *tr* which receive the training samples, *te* which receive the test samples, and *result* which receive 0 for class prediction and 1 for accuracy (for further details see *configBB.VarSel* method). For example, the code below could be used as a user-defined function using *logitboost*, and *random forest* as classifiers.

```
#EXAMPLE 1 : LOGITBOOST
#Install package boost from http://stat.ethz.ch/~dettling/boosting.html
library(boost)
#assuming 2-classes
logitboost.R.predict <- function (chr, parent, tr, te, result)
{
    d <- parent$data
    s <- logitboost(d$data[tr, chr], d$iclasses[tr]-1, d$data[te, chr])
    k <- ifelse(s[,ncol(s)] >= 0.5,2,1)
    if (result) sum(k == d$classes[te])/length(te)
    else k
}
bb.lb <- configBB.VarSel(..., classification.method="user",
classification.userFitnessFunc=logitboost.R.predict)

#EXAMPLE 2 : RANDOM FOREST
#Breiman, L. (2001), Random Forests, Machine Learning 45(1), 5-32
library(randomForest)
randomforest.R.predict <- function(chr, parent, tr, te, result) {
    d <- parent$data
    xrf <- randomForest(x=d$data[tr,chr], y=d$classes[tr],
            xtest=d$data[te,chr], ytest=d$classes[te])
    #if all(te==tr) resubstitution was specified, which is faster
    #considering that RF performs an internal cross-validation (out-of-bag)
    if (result) {
        if (all(te==tr)) sum(xrf$predicted==d$classes[te])/length(te)
        else sum(xrf$test$predicted==d$classes[te])/length(te)
    } else {
        if (all(te==tr)) xrf$predicted==d$classes[te]
        else xrf$test$predicted
    }
}
bb.lb <- configBB.VarSel(..., classification.method="user",
classification.userFitnessFunc=randomforest.R.predict,
classification.train.error="resubstitution")
```
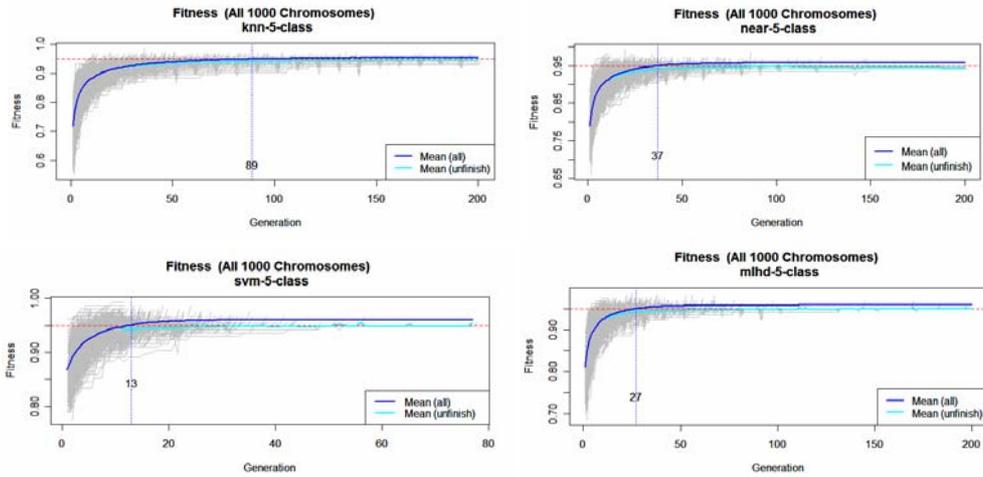
Figure 21 Comparison of fitness evolution in different classifiers.

3.3 Error Estimation

A very important issue in the development of a statistical model is the error estimation. There are several methods to estimate this quantity but they are all based on the fundamental principle that the most accurate procedure to estimate the degree of generality of a model is to asses the classification accuracy on a set of samples that has not been used to develop the model itself. Regarding this, classical approaches involve splitting data in training and test sets (see **BOX 1**). The training set is used to train the parameters of the classifier whereas the test set is left aside and it is used to asses the classification accuracy of the selected chromosomes at the end of a BigBang (see section 1.5). The fitness function uses the classification accuracy of the model on the training samples to assign a score to each chromosome and make possible the selection of better predictors through the law of natural selection. There are different ways to estimate the error during training. The most obvious one is to further split the training set into training and validation sets.
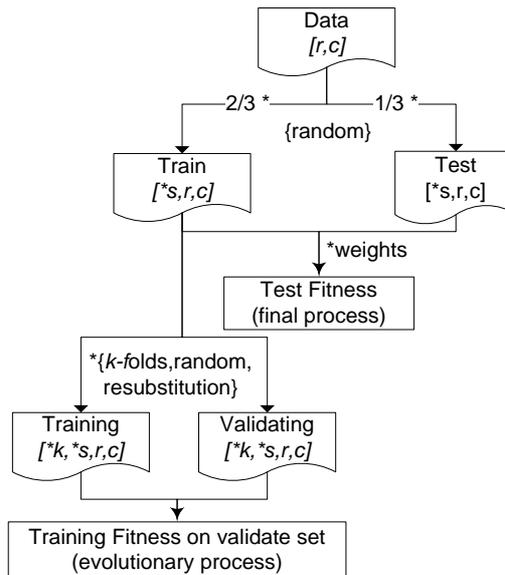
Figure 22 Error Estimation. The data is split *s* times randomly to generate train and test set proportional to 2/3 and 1/3 of samples respectively. *r* represent genes and *c* samples. These sets are used to compute the test error in the model selection process. Train set is split further into training and validating set to compute the train error in the evolutionary process (inside the Genetic Algorithm). All parameters marked with a star are configurable.

GALGO first splits the data, several times, into training and test sets (Figure 22). The second level consists in dividing each first-level training set into many second-level training and test sets, which is controlled by the *classification.train.error* parameter. The evolutionary process uses only these second-level sets remaining blind to the first-level test sets. This is controlled by the parameter *classification.mode*. GALGO chooses the appropriate data split according to the size of the training dataset according to the following formula:

```
# number of random splits is the number of samples
s = min(150, ncol(data))
# number of k-folds
k = round(13 - ncol(data)/11)
# but inside the range 3..number of samples
kfolds = max(min(k, ncol(data)), 3)
```

In our case study in the quick tutorial, 233 samples are divided into 156 and 77 samples in the first level (two thirds for training and one third for test, which is controlled by the *train* and *test* parameters). Then, in the second level, the 156 sample set is divided into three sets, all of which contain 52 samples. Thus, in the evolutionary process, the fitness function would use only 156 samples in three cycles (the fitness function would be called three times). Each cycle use one of the 52 sets as test samples and the remaining 104 samples as training. The splitting of samples is random and considers the

possible different number of samples per class, always trying to end up with the same class proportions than the original set. These behaviours and others can be customized through parameters (see reference manual for *configBB.VarSel* method). Finally, GALGO compute a large number of first-level splits in order to have a better estimation of the test error independent on the initial split. So, the final test error outside the *blast* process is estimated using the test set in all these first-level splits controlled by the parameter *classification.test.error*. For the data in the quick tutorial, the first-level split was done 150 times. Therefore, the test error estimation will call the fitness function 150 times using 156 samples for training and 77 samples for testing.

By default, the final error (outside the *blast* process) use only test samples for the estimation. However, this can be changed by using *classification.test.error* vector, which must contain 2 values interpreted as weights in training and test respectively. The error inside the evolutionary process (in *blast* or *evolve* methods for BigBang and Galgo objects respectively) can be computed using *"kfold"* (k-fold-cross-validation), *"loocv"* (leave-one-out-cross-validation), *"splits"* (random splits), and *"resubstitution"*. The number of folds in *"kfold"* is specified by *classification.train.Ksets* parameter and the percentage of splitting is controlled by *classification.train.splitFactor* parameter.

For example, let create a BigBang object using *configBB.VarSel* to estimate the final error as .632Bootstrap method (Ye 2003) as 0.632 weight for test and 0.368 for train, the train error as 10-fold-cross-validation, and the first-level splitting as 0.5 and 0.5 in only 20 splits.

```
bb <- configBB.VarSel(..., train=rep(0.5, 20),
test=rep(0.5,20), classification.test.error=c(0.368,
0.632), classification.train.error="kfolds",
classification.train.Ksets=10)
```

By default, the evolutionary process would evolve under the first split only. However, for example, if the user would like to change this behaviour selecting a different split in every BigBang cycle, the following code could be used.

```
bb <- configBB.VarSel(...)
bb$callPreFunc <- function(xbb, xgalgo) {
 xbb$data$selSplit <- sample(length(xbb$data$splitTrain))
  TRUE
}
```

Inside a BigBang object, *callPreFunc* will be called just before calling *evolve* of the GA (Galgo object).

Finally, the relevant variables for error estimation inside the BigBang object configured by *configBB.VarSel* method are:

selSplit	Selected split for training error evaluation. By default set to 1.
splitTest	A list containing vectors which specify the samples used as test. Each element in the list corresponds to one split.
splitTrain	A list containing vectors which specify the samples used as training. Each element in the list corresponds to one split.
splitTrainKFold	A list containing lists which specify the samples used as training. Each element in the list corresponds to a list that contains the samples used for training in each *"k-fold"*, which are used for all partitioning schemes in *classification.train.error*.
splitValidKFold	Similar to *splitTrainKFold* but containing the validating (test) cases.
testErrorWeights	Vector containing the weights for train and test to estimate the final error.
splitAll	List containing vectors which contain the samples used in that particular split (*splitTest*+*splitTrain*).

3.4   BigBang Object Configuration

The BigBang object is composed of methods and properties (variables). The most important methods are the plot method, which were explained in sections 2.4, 2.5, 5, and 6, and the *blast* method. In this section we will explain, in general terms, how the blast process works and what is the meaning of variables included in the BigBang object.

3.4.1   Blast Process

1. Initialize Timings
2. While maxBigBangs and maxSolutions has not been reach do
    2.1. Initialize Galgo and generate new random population
    2.2. If the result from calling callPreFunc is NA then exit while
    2.3. Evolve Galgo
    2.4. Save Formatted Best Chromosome from Galgo
    2.5. If the result from calling callEnhancerFunc using the best chromosome is not NULL
        2.5.1.   Save original chromosome in evolvedChormosomes
        2.5.2.   Replace Best Chromosome with the result of callEnhancerFunc
    2.6. If the result from calling callBackFunc is NA then exit while
    2.7. Increase current bigbang and solutions (if any)
    2.8. If saveFrequency then call saveObject
    2.9. If gcFrequency then call gc
3. Call saveObject

4. End

3.4.2 Variables in BigBang Object

bb	Numeric. Current BigBang cycle
bestChromosomes	List of formatted evolved or refined chromosomes.
bestFitness	List of numeric values with chromosome fitness.
call	The original call made to create the BigBang object.
callBackFunc	Function to be called after the galgo evolution. If the result is NA, the *blast* process will end.
callEnhancerFunc	Function to be called after the galgo evolution. If the result is not NULL, it is assumed it represent a refined chromosome, which is saved in bestChromosomes.
callPreFunc	Function to be called before the galgo evolution. If the result is NULL, the process will end.
classes	Class of the samples (if any).
collectMode	Type of object to be saved (see BigBang method).
data	Any user data. *configBB.VarSel* methods store here the data, the splits and many other values.
elapsedRun	Elapsed time of galgo process in seconds.
elapsedTime	Elapsed time of the whole process.
evolvedChromosomes	When *callEnhancerFunc* is provided, stores the original chromosomes.
evolvedFitness	When *callEnhancerFunc* is provided, stores the original fitness.
galgo	Galgo object to be evolved. This should be properly configured. *configBB.VarSel* methods configure it depending on the parameters.
galgos	If *collectMode=="galgos"* it stores the galgo object for each evolution.
gcCalls	Number of calls to garbage collector.
gcFrequency	How often the garbage collector will be called.
geneNames	Names of the genes.
generation	Numeric vector with the last generation of galgo evolutions.
iclasses	Integer representation of sample classes.
id	Id of the BigBang object.
leftTime	Remaining Time.
levels	Levels of sample classes.
main	User title for plots.
maxBigBang	Maximum number of BigBang cycles.
maxCounts	Resolution of the *generankstability* plot.
maxFitnesses	List of numeric values with the evolution of the best fitness across generations in every galgo.
nClasses	Total number of classes.
onlySolutions	Specify if only solutions reached should be saved.
running	Flag indicating if the BigBang is inside the *blast* process.
sampleNames	Names of samples.

saveFile	Name of the file to save the BigBang object.
saveFrequency	How often (in BigBang cycles) the file should be updated.
saveGeneBreaks	Used to cut values of genes when they are continuous. For variable selection problems, it contains the integer intervals.
saveMode	Type of saving.
saveVariableName	The name of the variable that will be used to save the object.
solution	Logical vector indicating whether the galgo reach solution.
solutions	The total number of solutions saved.
startedTime	The start time.
timing	Numeric vector indicating the elapsed seconds in each galgo evolution.
userCancelled	Flag indicating that the user cancelled the process through callBack mechanism.
verbose	The level of messaging.
(any other variable)	User variable or temporal/cache variable used to speed up processes.

3.4.3 Chromosome Formatting Scheme

The BigBang object stores the chromosomes formatted instead of the original chromosome object. This resulted in memory and performance improvements when the number of chromosomes to save is large. By default, the chromosomes are converted to a numeric vector and saved. However, if the user needs to save the chromosomes in the original object or any other format, it can be attain overwriting the *formatChromosome* method, using the following code.

```
setMethodS3("formatChromosome", "BigBang",
function(.bbO, chr, ...) {
    chr
})
```

3.5 $data Configuration

The *configBB.VarSel* methods create the data variable in the BigBang object. In the error estimation section we showed the variables created regarding the error estimation scheme. The table below shows the others variables created.

classes	Factor. The samples classes.
classFunc	Function that predict the class for a given chromosome. The default is *classPrediction* method.
classificationMethod	The name of the classification method.
data	The **transposed** data.
fitnessFunc	Function that computes the accuracy of a given chromosome. The default is *fitness* method.

iclasses	Integer representation of classes.
modelSelectionFunc	Function to be used to compute the generalized final error. It depends on splits and set. The default is *modelSelection* method.
predictFunc	Function to compute the accuracy or class prediction of test samples based on training samples and a chromosome. The default depends on the method. It can be *mlhd.R.predict, mlhd.C.predict, knn.R.predict, knn.C.predict, nearcent.R.predict, nearcent.C.predict, svm.C.predict, svm.R.predict, nnet.R.predict,* or *rpart.R.predict.*
scale	Logical. Describe if scale was used (mean=0, variance=1).

3.6   Genetic Algorithm Configuration (Galgo Object)

3.6.1   *evolve* Process

1. Initialize generation=0
2. While maxGeneration has not been reach
    2.1. *evaluate* all chromosome populations
    2.2. Save the best chromosome
    2.3. If the result of *callBackFunc* is NA then exit while
    2.4. If goalFitness has been reach then exit while
    2.5. Generate *progeny* for all chromosome populations doing the following
        2.5.1.   Call *migrate* (for world objects)
        2.5.2.   Generate *offspring* by "natural" selection
        2.5.3.   Perform *crossover*
        2.5.4.   Call *mutate*
3. End

3.6.2   Fitness Function

The default fitness function is *fitness* (type ?fitness in R) which is defined as:

```
fitness <- function(chr, parent) {
    d <- parent$data
    s  <- 0
    tr <- d$splitTrainKFold[[d$selSplit]]
    va <- d$splitValidKFold[[d$selSplit]]
    for (k in 1:length(tr))
        s <- s + d$predictFunc(as.numeric(chr), parent,
    tr[[k]], va[[k]], as.integer(1))
    s/length(tr)
}
```

parent, in the above code, is assumed to be the "blasting" BigBang object. Note the usage of the variables inside *parent$data*, *splitValidKFold* and *splitTrainKFold*, which contain the second-level train and validation (test) sets used to estimate the error (see "$data Configuration" section). Thus, this function returns the average test error in all the second-level validation folds. That is, the fitness in the evolutionary process is **completely blind** to the first-level test set. Different criteria can be implemented overwriting this function, simply by assigning a new function to *fitness*. *predictFunc* contains the specific function needed for the classification method of our choice. Thus, *predictFunc*, as shown in the "$data Configuration" section above, can be any of *mlhd.R.predict*, *mlhd.C.predict*, *knn.R.predict*, *knn.C.predict*, *nearcent.R.predict*, *nearcent.C.predict*, *svm.C.predict*, *svm.R.predict*, *nnet.R.predict*, *rpart.R.predict*, or a user function with similar parameters (see User-Specific section above).

3.6.3 Offspring

The *scaling* method (in Niche object) converts the fitness value to a weight or probability for the chromosomes to be selected to generate progeny for the next generation. By default, the model used is:

```
p = scaleFac * max(0,fitness - meanFac * mean(fitness))^powFac
```

The previous model generates large values for fitness above the mean and small values for fitness below the mean. The default values used by *configBB.VarSel* are 1, 0.85, and 2 for *offspringScaleFactor*, *offspringMeanFactor*, and *offspringPowerFactor* respectively.

To change this scaling mechanism, the user needs to overwrite the *scaling* method in Niche before creating any Niche object, thus before calling *configBB.VarSel*. The code below is an example.

```
setMethodS3("scaling", "Niche", function(ni, ...) {
    # code to return a weighted vector
    # which must depend on ni$fitness
})
```

The default *offspring* method use random generated number weighted using the *scaling* method to select the chromosomes to be replicated. The length of population of chromosomes is static. To change the *offspring* mechanism, just replace the *offspring* method in the Niche object similarly to the replacement of the *scaling* method shown above.

3.6.4 Crossover

The default *crossover* method performs 0.5*n* crossovers where *n* is the population size of the Niche, which is the result of calling *crossoverFunc* variable stored in the Niche object. For this, it uses the *crossoverPoints* variable which *configBB.VarSel* set to 1 at the middle of the chromosome. These can be changed using *crossoverFunc* and *crossoverPoints* parameter in *configBB.VarSel.* The other possibility is to replace the *crossover* method in the Niche object as in the *offspring* example shown above.

3.6.5 Mutation

The default *mutation* method performs *n* mutations where *n* is the population size of the Niche, which is the result of calling *mutationFunc* variable stored in the Niche object. These can be changed using *mutationFunc* parameter in *configBB.VarSel.* The other possibility is to replace the *mutate* method in the Niche object as in the *offspring* example shown above.

3.6.6 Elitism

The default *elitism* is a probability vector, which contains 1 nine times and 0.5 1 time. This means elitism on in 9 consecutives generations followed but elitism with probability 0.5. Generation 11 to 20 behave similar to generation 1 to 10 (it is a circular vector). Elitism can be changed to a fixed value or to a function in the *elitism* parameter in *configBB.VarSel*. Elitism has a great impact in the search (see later sections).

3.6.7 Migration

The default migration (*immigration*) is a probability vector, which contains 0 18 times, 0.5 once, and 1 once. As *elitism*, *immigration* values can be a fixed value or a function. However, *elitism* applies to a Niche objects whereas *immigration* applies for *World* object instead. Anyway, migration is turned off because by default the number of niches is 1.

3.7 Setting-up non-classification problems

configBB.VarSel assumes that a classification problem is being set up, there is however, a *configBB.VarSelMisc* method, which does not make this assumption. The major difference is that the fitness function must always be provided. The fitness function must follow the prototype shown in the fitness section and could use the splitting scheme created similar to the following code.

```
my.fitness <- function(chr, parent) {
    d <- parent$data
    s  <- 0
    tr <- d$splitTrainKFold[[d$selSplit]]
    va <- d$splitValidKFold[[d$selSplit]]
    for (k in 1:length(tr))
        s <- s + fitting.prediction(as.numeric(chr),
    parent, tr[[k]], va[[k]])
    s/length(tr)
}
```

In this code, *fitting.predicion* is the function that makes the user computations depending on the chromosome, the *parent* BigBang object, a set of training samples to learn the model parameters, and a set of testing samples to asses the fitting. However, the fitness function coding is up to the user because the only requirement is that *my.fitness*, in this case, returns a value representative of how good the chromosome *chr* is in its appropriate context. In this sense, the user can really execute any sensible code, which may or may not follow the structure shown above.

3.8   Setting-Up the Analysis Manually

To set-up the analysis we have to create a BigBang object, which contain a prototype Galgo object that would be used to evolve models resulting in chromosomes to be stored for further analysis. In turn, the Galgo object needs the fitness function, populations of chromosomes, and GA parameters. The population of chromosomes can be a list of Niches or World objects. Niches contain Chromosomes objects, which in turn contain Gene objects. *configBB.VarSel* and *configBB.VarSelMisc* wrapper methods will configure all these objects in order to facilitate common tasks. However, these methods do not consider all possible scenarios. The task of creating all objects is in some how, tedious and perhaps complex. To avoid all the work needed, in some cases, we could use these wrappers to build a prototype BigBang object, then replacing the variables inside the object as desired. Other option is creating all objects by hand. To illustrate this process, we will use pseudo-code similar to those implemented in the wrappers functions.

First we must create (and print) Gene objects as the following examples (see Gene object typing ?Gene in R).

```
gen1 <- Gene(shape1=1, shape2=100,
generateFunc=function(g,n,sh1,sh2) runif(n,sh1,sh2))
gen1
```

or

```
gen2 <- Gene(shape1=0, shape2=1, generateFunc=
function(g,n,sh1,sh2) rnorm(n,sh1,sh2))
gen2
```

Then, we create a chromosome prototype object.

```
chr1 <- Chromosome(genes = newCollection(gen1, 5))
chr1
```

The code above creates a Chromosome object that contains the same gene prototype (different gene objects though). However, a Chromosome may contain different gene prototypes.

```
chr2 <- Chromosome(genes =
list(clone(gen1),clone(gen2),clone(gen1),clone(gen1)))
chr2
```

clone is important because we need to add an independent new instance of that object.

Now, we can create a Niche object from chromosomes as in the following code.

```
niche1 <- Niche(chromosomes = newRandomCollection(chr1,
50))
niche1
```

Because Galgo object can operate using Niche or World objects, the creation of World object is optional. Let create one using the following code.

```
world1 <- World(niches = newRandomCollection(niche, 3),
immigration=0.01)
```

We can create now, the Galgo object using a code similar to the following.

```
galgo1 <- Galgo(
        populations=newRandomCollection(world1, 1),
        goalFitness=0.666,
        minGenerations=50,
        maxGenerations=500,
        verbose=1,
        fitnessFunc=???)
```

So far, we have created a Galgo object which can be used to evolve solutions. We will need to build a BigBang object if and only if we need to analyze several solutions and we would like to acquire solutions automatically. Otherwise, the Galgo object created would be enough to get an individual solution. For example using the following code.

```
> evolve(galgo1)
> best(galgo1)
```

Finally, the BigBang object can be created as in the following code.

```
bigbang <- BigBang(galgo=galgo1,
    maxBigBangs=100000000,
    maxSolutions=99999999,
    saveFile="manual-blabla.Rdata",
    saveVariableName="bb.manual",
    data=...,
    main="manual bb")
```

3.9 Extending Objects

In object oriented programming (OOP), an important issue is the creation of sub-classes of previously defined objects, commonly referred as *extending* or *inheriting* object definition. In GALGO, objects can be extended very easily, for instance let extend the Gene object then create chromosomes that consist on this objects instead using the following code.

```
#define a new object
setConstructorS3("MyGene", function(myValue=0,...) {
    extend(Gene(...), "MyGene", myValue=myValue)
})

#create a new mutate operator
setMethodS3("mutate", "MyGene", function(object, ...) {
})

gen3 <- MyGene(shape1=0, shape2=10, myValue=2)
class(gen3)
chr3 <- Chromosome( genes=newCollection(gen3,3) )
chr3
```

The code above is only illustrative, the Gene and any other object can save any user variable. However, this example illustrates how different objects can be built and extended preserving the expected behaviour (of Chromosome in this case).

3.10 Summary

We have reviewed several options to set-up an analysis that can be more convenient for a given problem. These options range from simple changes in parameter configurations in *configBB.VarSel* to creating the BigBang object entirely by hand. We have seen that GALGO can be configured to non-classification problems relatively easy.

4 Step 2 - Evolving Models / Chromosomes

The evolving step is the more time consuming part of the process. Our aim in this section is to understand the output generated and how to change it for our own purposes. We will assume that *configBB.VarSel* has been used to configure the BigBang object.

4.1 Outputs

The default output shown below from running the *blast* method really consists on two contiguous outputs. Lines starting with [Bb] meant the output of the *blast* process for the BigBang object. Lines starting with [e] meant the output of the *evolve* method in the galgo object inside the BigBang. This output is controlled by *bigbangVerbose* and *galgoVerbose* parameters (type ?configBB.VarSel, ?Galgo and ?BigBang in R), which can be cancelled by assigning 0.

```
[Bb] Starting, Solutions=300
[Bb]    #bb     Sol     Last    Fitness %Fit    Gen     Time    Elapsed
Total   Remaining
[e] Starting: Fitness Goal=0.9, Generations=(10 : 200)
[e]     Elapsed Time    Generation      Fitness %Fit    [Next Generations]
[e]     0h 0m 0s        (m)     0       0.64103 71.23%  +++++++...+.........
[e]     0h 0m 6s                20      0.87179 96.87%  ....................
[e]     0h 0m 14s               40      0.87179 96.87%  .....+..+.....+.+...
[e]     0h 0m 22s               60      0.92308 102.56% +
[e]     0h 0m 22s       ***     61      0.92308 102.56% FINISH: 2164 1612...
[Bb]    300     299     Sol Ok  0.92308 102.56% 61      22.16s  3722s
4054s   14 (0h 0m 14s )
```

Looking at the first line and second lines of each output, the values are self-explained. For the section [Next Generations] a "+" means that the maximum fitness has increased, "-" means that has decreased, "." means that has no changed, and "G" means that the fitness goal has been reach but a termination criteria has not been fired. See section 2.3 for details.

The default graphical output is the plot of three "diagnostic" plots as shown in Figure 7 (see section 2.3 for details). This output can be changed specifying the parameter *callBackFuncBB* as below.

```
bb <- configBB.VarSel(...,callBackFuncBB=function(xbb,
galgo) { plot(xbb,type=c("fitness","generankstability"),
mord=100)} )
```

In general, *callBackFuncBB* can be any code computing any desired computation. For the graphical Galgo output shown in Figure 8 the options

are more limited, however, section 2.3 explains how to activate it using *callBackFuncGALGO* parameter. Again, using this parameter, in general, the user can perform any kind of computation.

4.2   Process Interruption

As explained in section 2.3, the *blast* process can be interrupted by pressing Esc in windows or Ctrl-C in "Unix" to perform a pre-analysis and then resumed later using the *blast* method again.

4.3   Adding and Merging Solutions

Sometimes we end up with a number of solutions which are not sufficient in which case we could use the *blast* method specifying the *add* parameter to force to search for more solutions. Other situation arises when we have performed several isolated GALGO searches and we would like to "merge" them and perform a "joint" analysis. In this case one could use the method "mergeBangs" (type ?mergeBangs.BigBang in R) to merge chromosomes from different BigBang objects. Another situation is when we have a certain number of solutions which we believe are not enough for "rank stability" and we would like to know more precisely how many solutions we need to get more stable ranks. To have an idea, we can duplicate randomly previous solutions and see how the "generankstability" plot behaves. Once we have added enough solutions, we have an estimate of the total number of solution needed. Of course this is not ideal because we are duplicating original solutions, but at least we can have a very good estimate in few seconds. To do this, use the *addRandomSolutions* method (see method description typing ?addRandomSolutions.BigBang).

5   Step 3 - Analysis and Refinement of Chromosome Populations

Most of the plots shown here are sensitive to the number of desired genes to analyse, the *mord* parameter, and the chromosome population to analyze, the *filter* and *subset* parameter. The gene colouring is defined by *rcol* and *mcol* parameters. *mcol* is the number of "automatic" colours, and *rcol* are the absolute colours (type ?plot.BigBang for details).

5.1   Analysis of Gene Frequencies and Ranks

In this section we will review the plots and methods implemented in GALGO regarding the occurrence of gene in models.

5.1.1   Gene Frequency

Shows the number of times (in vertical axis) a gene (or variable, in horizontal axis) has been present in a population of chromosomes. By default, the 50 most frequent genes are coloured in 8 different colours (with slightly different intensities). Figure 23 shows the gene frequency for all genes colouring the 100 most frequent genes (*mord*=100).

```
> plot(bb.nc, type="geneFrequency", mord=100)
> f <- geneFrequency(bb.nc) #as a numeric table
> f
```
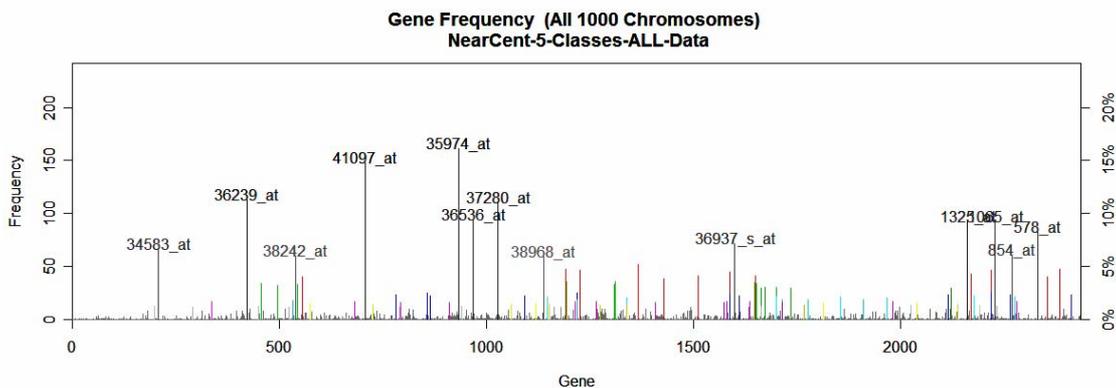

Figure 23 Gene Frequency. Left axis shows the number of times the gene has been present in chromosomes. Right axis marks the corresponding percentage relative to the total number of chromosomes. Horizontal axis shows the value of the gene in the chromosome. Only the first "colour section" is labelled with their corresponding gene names.

As an example, let compute the gene frequency of the "better models" from the training fitness, as follows.

```
> bf <- unlist(bb.nc$bestFitness)
```

```
> better <- bf > median(bf)
> plot(bigbang, type="geneFrequency", subset=better)
> fb <- geneFrequency(bb.nc, subset=better)
> plot(as.numeric(f), as.numeric(fb), pch=20, main="Freq
vs Freq(better)")
```

Figure 24 shows that the frequency of the genes in all solutions is very high correlated with the gene frequency of the better models, which suggest, roughly, that the better models do not have preference in genes (in this population of chromosomes).
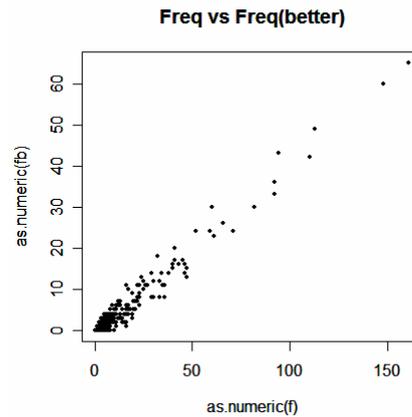
Figure 24 Gene Frequency Comparison.

5.1.2 Gene Ranks

Shows the frequency (vertical axis) of most frequent genes ordered by rank (horizontal axis).

```
> plot(bb.nc, type="generank")
> r <- geneFrequency(bb.nc, value="rank")
> which(r == 1)
```
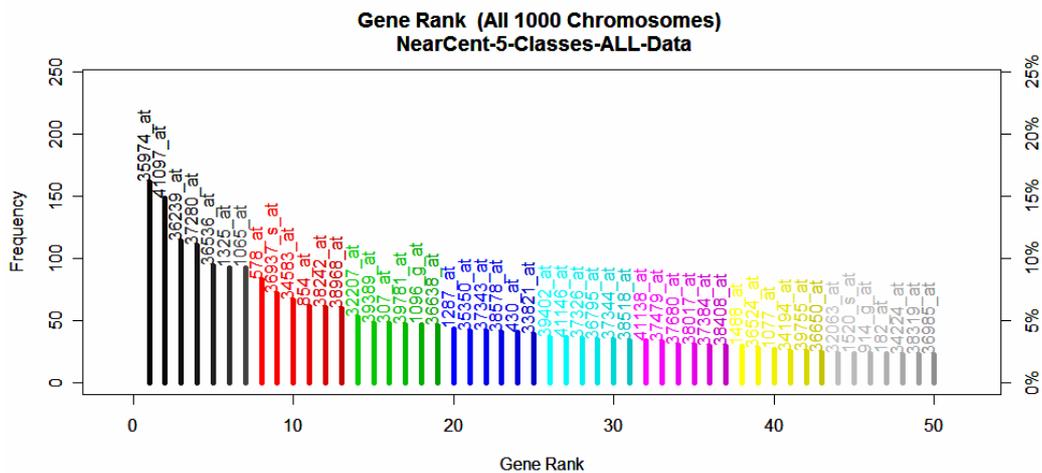

Figure 25 Gene Rank.

5.1.3 Stability of Gene Ranks

Because we have assumed that the gene frequency is related to the importance (Li *et al.* 2001), it is important to check if the ranks are reasonable stable. We have designed a plot for showing the rank "changes" when solutions were being evolved in vertical axis. The *n* most frequent genes are shown in horizontal axis sorted by rank. When genes have many changes in ranks, the plot displays different colours; hence several changes in colours (and which specific colours) indicates instability. Commonly the top 7 "black" genes are stabilized quickly, in 100 to 300 solutions, whereas low ranked "grey" genes would require thousands of solutions to be stabilized. The fact that the ranks are stable is in somehow important, because if ranks are affected by random variations, similar runs would end up in different ranks, in which case the chromosomes are not representative of the entire population of possible solutions.

```
> plot(bb.nc, type="generankstability")
```
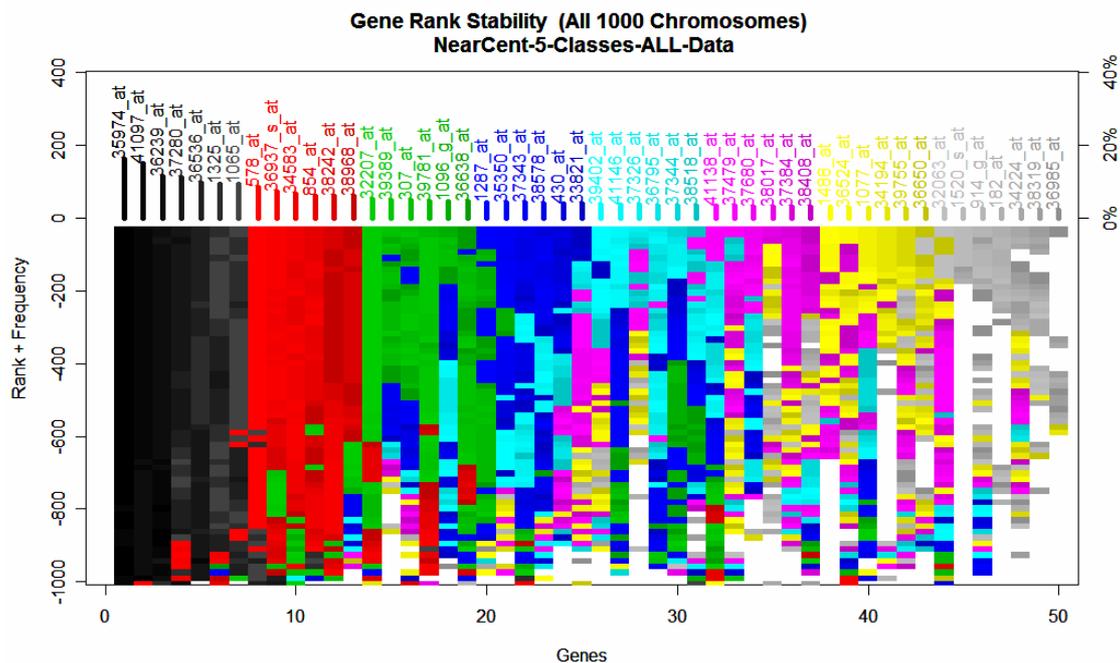

Figure 26 Stability of Gene Ranks.

5.1.4 Rank Index

This plot shows the ranks of all genes (horizontal axis) and their frequency in log scale (in vertical axis) to highlights small frequencies and compact high frequencies.

```
> plot(bb.nc, type="rankindex", cex=.9)
```
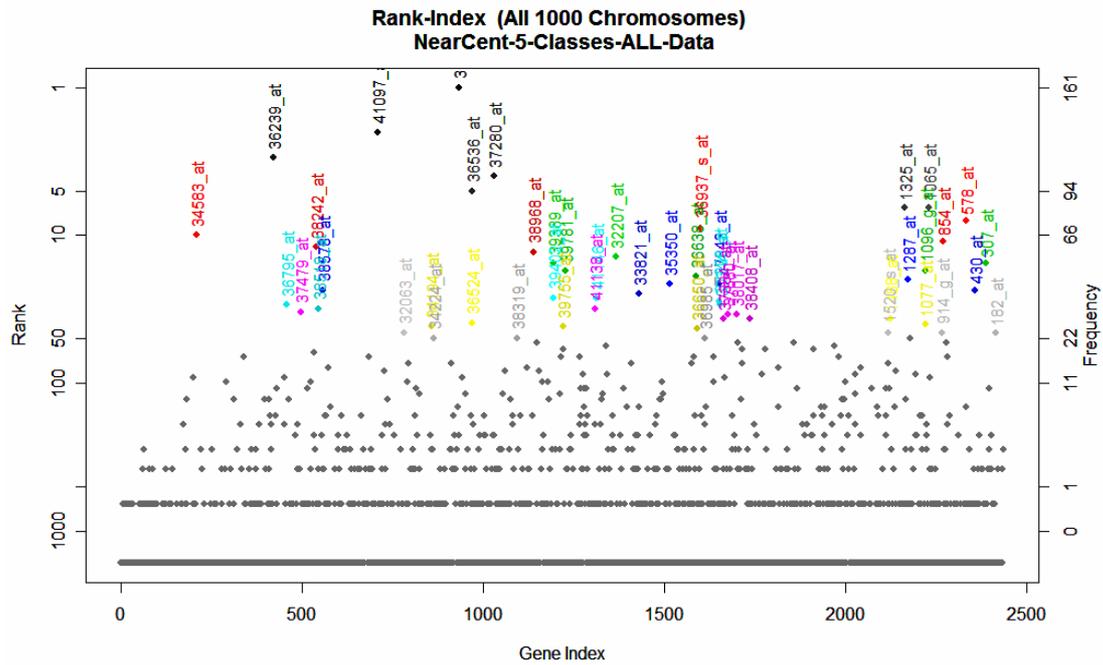

Figure 27 Rank Index.

5.1.5 Gene Frequency Distribution

Figure 28 shows the distribution of gene frequencies in horizontal axis and the number of genes, in log-scaled vertical axis, with that particular gene frequency.

```
> plot(bb.nc, type="genefrequencydist")
```
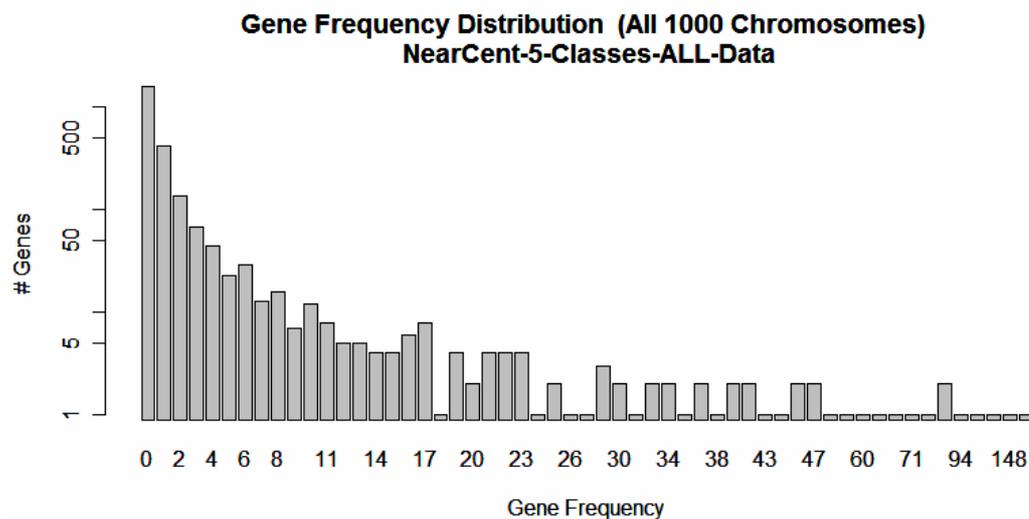

Figure 28 Distribution of the gene frequency. Vertical scale is logarithmic.

5.1.6 Number of Genes and Frequency

Figure 29 shows the number of genes that have frequency higher that any frequency (similar to a cumulative distribution function). For example, there are approximately 55 genes with frequency higher than 20 (marked with a vertical dotted line).

```
> plot(bb.nc, type="topgenenumber")
> abline(v=20, lty=3, col="grey")
```
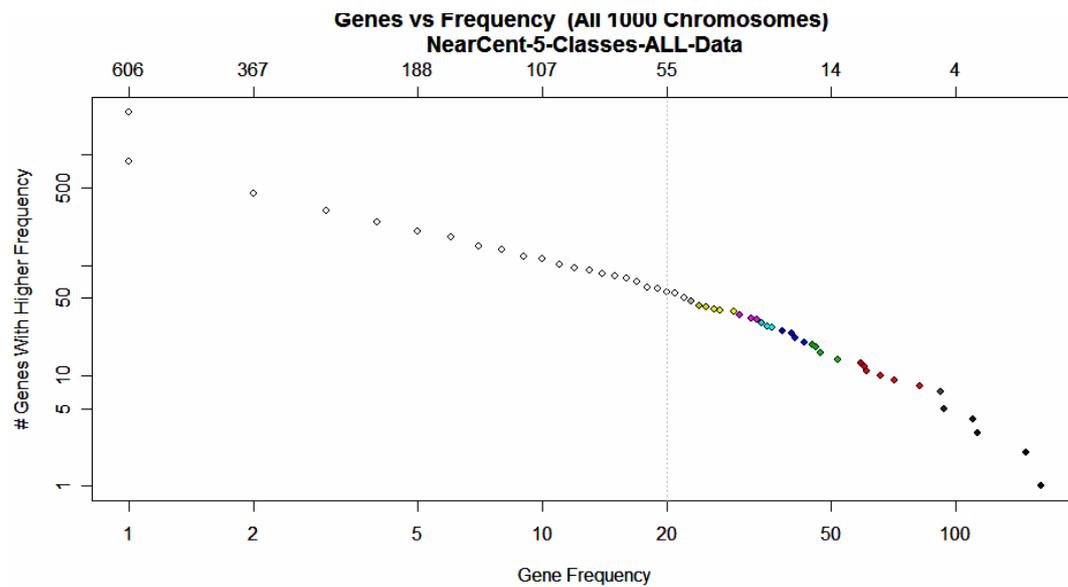

Figure 29 Genes higher than any frequency. Bottom horizontal axis is gene frequency in logarithmic scale. Top horizontal axis is the approximate rank for marked frequencies which should be similar to the number of genes in vertical axis.

5.1.7 Top Ranked Genes Used by Models

Because we would use top-ranked genes for generating models (see section 6), one important decision is how many genes to use as "top-ranked genes". One factor to make decision could be how many genes are actually present in the chromosome population. Figure 30 shows that 416 of the total of 2435 genes in the dataset are present in the example ran here. Now, because there are genes that appear frequently in chromosomes, Figure 30 shows the number of genes present in all chromosomes and the number of top-genes needed to cover fractions of them. This could help to decide how many genes to select covering more or less genes in chromosomes as criteria. For instance, the top 14 genes are the 25% of all different genes present in chromosomes. This is because these genes are highly repetitive in chromosomes.

```
> plot(bb.nc, type="genecoverage")
```
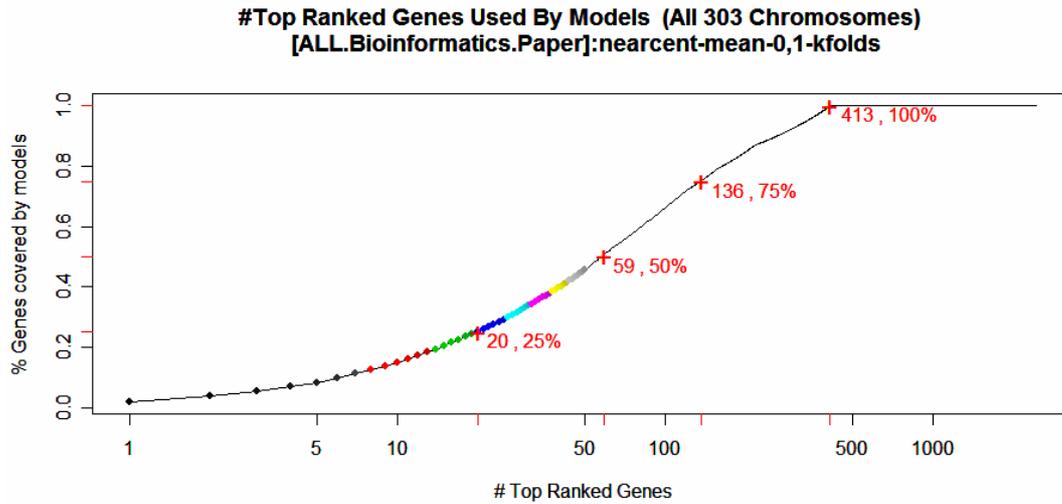
Figure 30 Number of Top Ranked Genes Used by Models.

5.1.8 Top Ranked Genes in Models

```
> plot(bb.nc, type="genesintop")
> abline(v=log10(100), lty=3, col="grey")
```
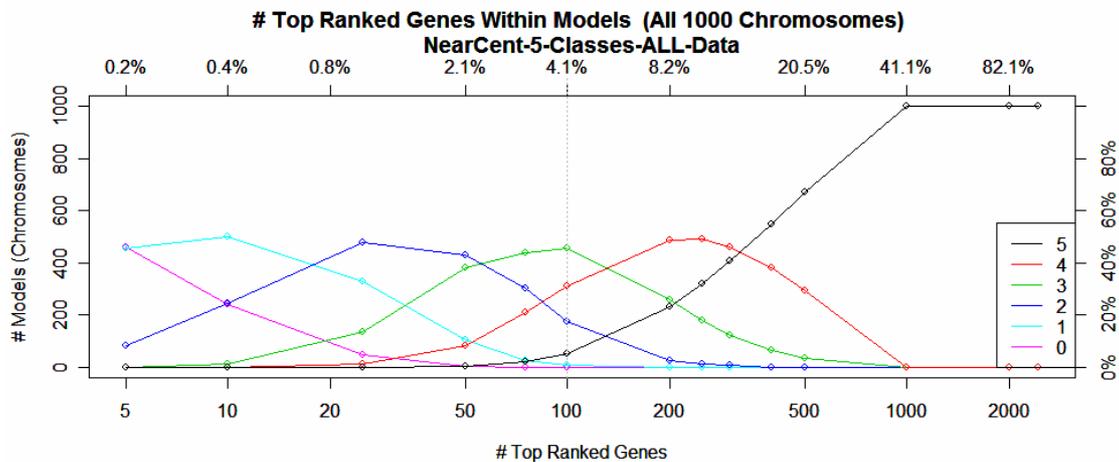

Figure 31 Distribution of the Number of Top-Ranked Genes within Chromosomes.

For example, using 100 top-ranked genes, there are approximately 50% of models that contains 3 of these 100 genes, 30% containing 4 genes, ~17% having 2 genes, and ~4% with 5 genes (Figure 31). That is, if we use 100 top genes and remove all other genes from the chromosomes, these values are the result of the distribution of these shrunken chromosomes.

5.2 Analysis of Models

5.2.1 Overlapped Genes in Models

Figure 32 shows the composition of the models in terms of top-ranked genes. By default, the chromosomes are sorted by its most top-ranked genes; hence, chromosomes with similar top-ranked-genes are stacked together. Chromosomes are shown in vertical and genes in horizontal. For example, we can see easily which genes has been combined with the first top-gene.

```
> plot(bb.nc, type="geneoverlap",cex=.75)
```
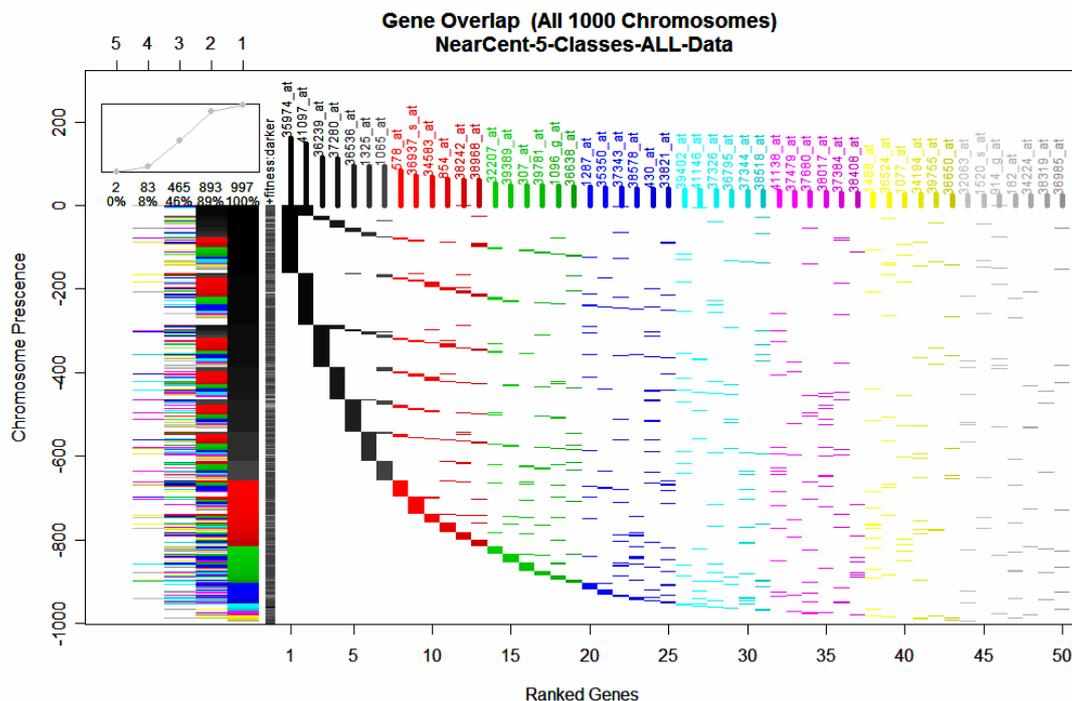

Figure 32 Overlapped Genes in Models.

5.2.2 Gene Interaction-Network

Figure 33 shows the dependency of top-ranked genes with each other. The line thickness represents the dependency strength relative to the population of relations shown. By default, only the two most important dependencies per gene are shown.

```
> plot(bb.nc, type="genenetwork")
```
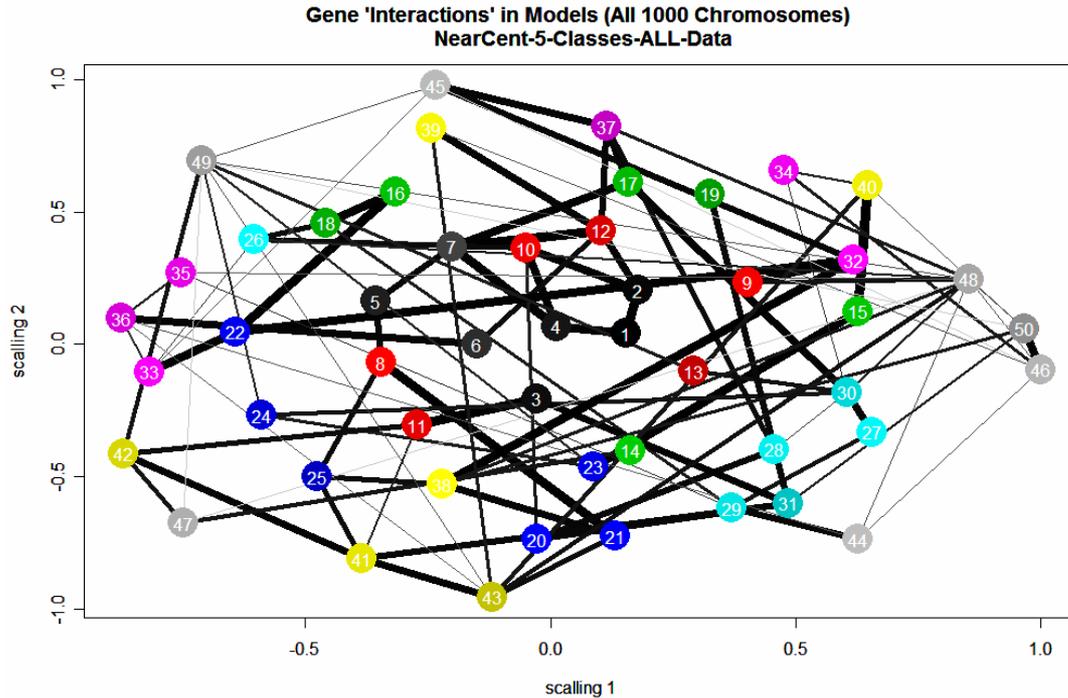
Figure 33 Gene Interactions within models.

5.3   *Analysis of Model Accuracies*

5.3.1   Confusion Matrix

For each class and each sample, shows the average percentage of the times that sample or class has been predicted as any class. By default, the error is computed in the test set for all splits (often, the number of splits is the number of samples); hence, by default, a sample is approximately predicted P number of times, where P = # chromosomes * number of splits * testfactor, and test factor is, by default in *configBB.VarSel*, 1/3. For basic details see section 2.4.2.

```
> plot(bb.nc, type="confusion")
> cpm <- classPredictionMatrix(bb.nc)
> cpm
> cm <- confusionMatrix(bb.nc, cpm)
> cm
> sensitivityClass(bb.nc, cm)
> specificityClass(bb.nc, cm)
> length(bb.nc$bestChromosomes) # number of chromosomes
(see plot titles)
> length(bb.nc$data$splitTrain) # number of splits
#~150
> len.test <- length(bb.nc$data$splitTest[[1]])
> len.train <- length(bb.nc$data$splitTrain[[1]])
> len.test/(len.test+len.train) # test-factor
#~0.33 ~ 1/3
```
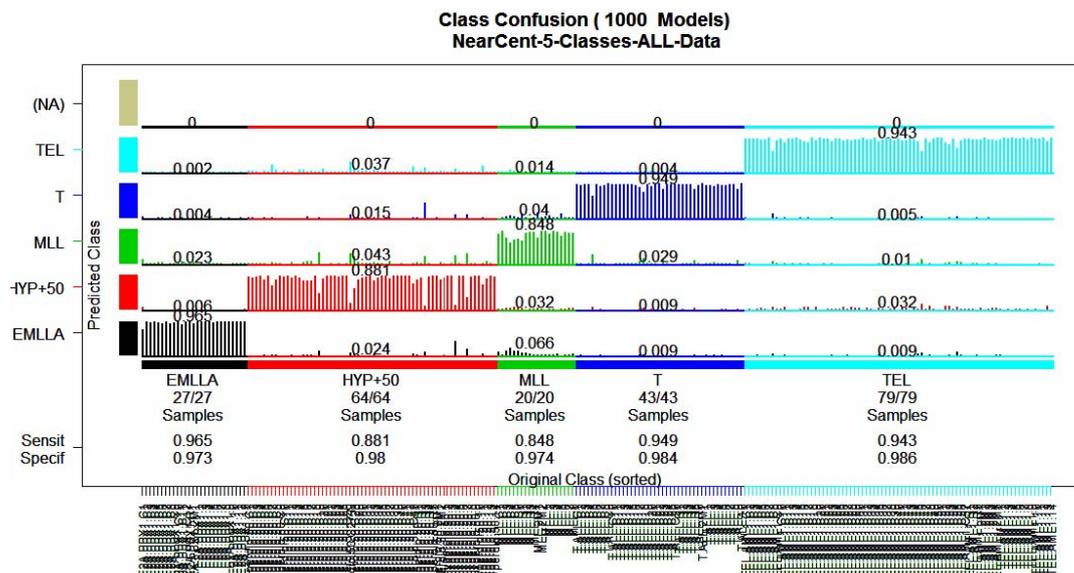
Figure 34 Confusion Matrix. All samples sorted by known class in horizontal. Vertical shows the fraction of times any given sample has been predicted as any other class.

To assess the error in training or test sets and in different splits, we can use the following code.

```
> plot(bb.nc, type="confusion", set=c(1,0), splits=1:10)
```

The code above will evaluate all chromosomes only in training using the first 10 splits. *set* and *splits* are really parameters passed to *classPredictionMatrix* method used in to draw the plot. *classPredictionMatrix* does not use these parameters, instead, they are passed to *bb.nc$data$classFunc*, which was set by *configBB.VarSel* equal to *classPrediction* method (see section 3.5 and type ?classPrediction). It is just there where *set* and *splits* have a real processing. Therefore, these parameters can be used also in the *classPredictionMatrix* method.

```
> classPredictionMatrix(bb.nc, set=c(1,0), splits=1:10)
```

A more compressed format is shown in Figure 35, which highlights the distributions of the averages of class prediction for each class. This plot was produced using the following code.

```
> plot(bb.nc, type="confusionbox")
```
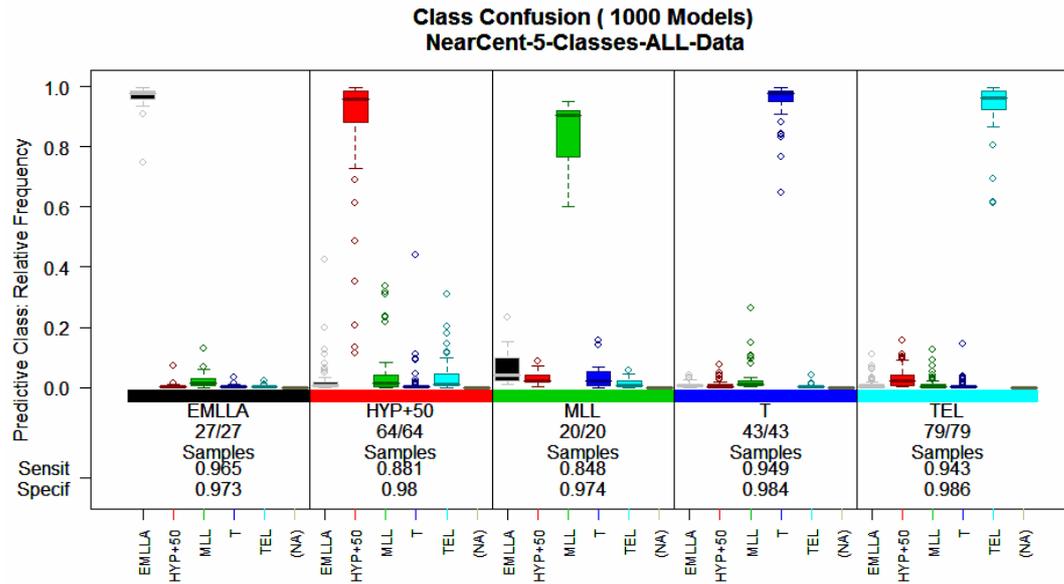
Figure 35 Confusion matrix by class prediction distribution.

A third format can be produced using the following code.

```
> plot(bb.nc, type="confusionpamr")
```
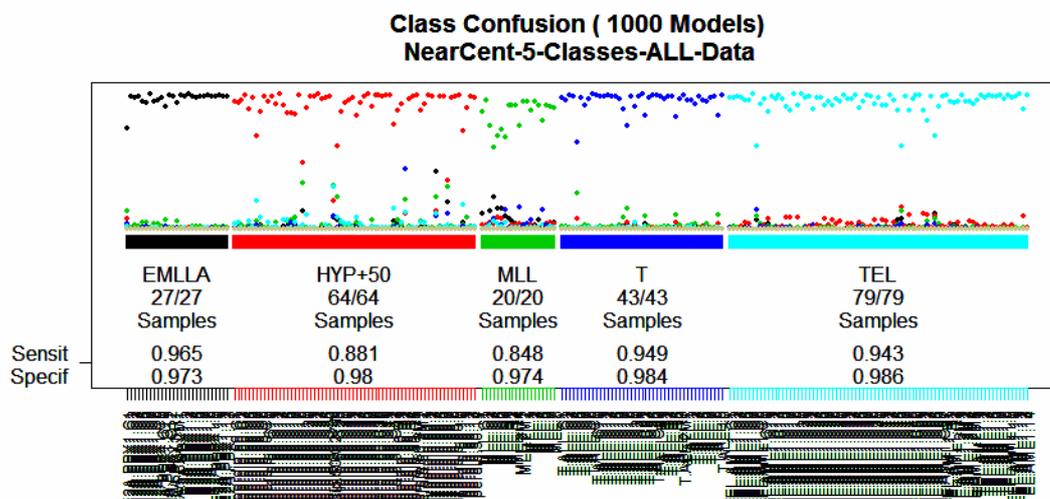

Figure 36 Confusion Matrix in a single row.

Finally, as was seen in section 2.4.2, the accuracy for a specific chromosome can be assessed as follows.

```
> plot(bb.nc, type="confusion",
chromosomes=list(bb.nc$bestChromosomes[[1]]))
```

5.3.2 Chromosome Accuracies

```
> plot(bb.nc, type="splitsfitness", xlab="Split (TEST-
SETS)")
> fs <- fitnessSplits(bb.nc)
> dim(fs) #~ 303 x 150
> fs
> plot(bb.nc, type="splitsfitness", set=c(1,0),
xlab="Split (TRAINING-SETS)")
```

Figure 37 shows the distribution of the fitness across all splits. By default, the fitness is computed in all test sets. Again, as in the sections above, using *set* parameter, which effects this time to *fitnessSplits*, we could assess the error in training. To evaluate only a set of chromosomes, we can use the *filter* or the *subset* parameter or use directly the *chromosomes* parameter.
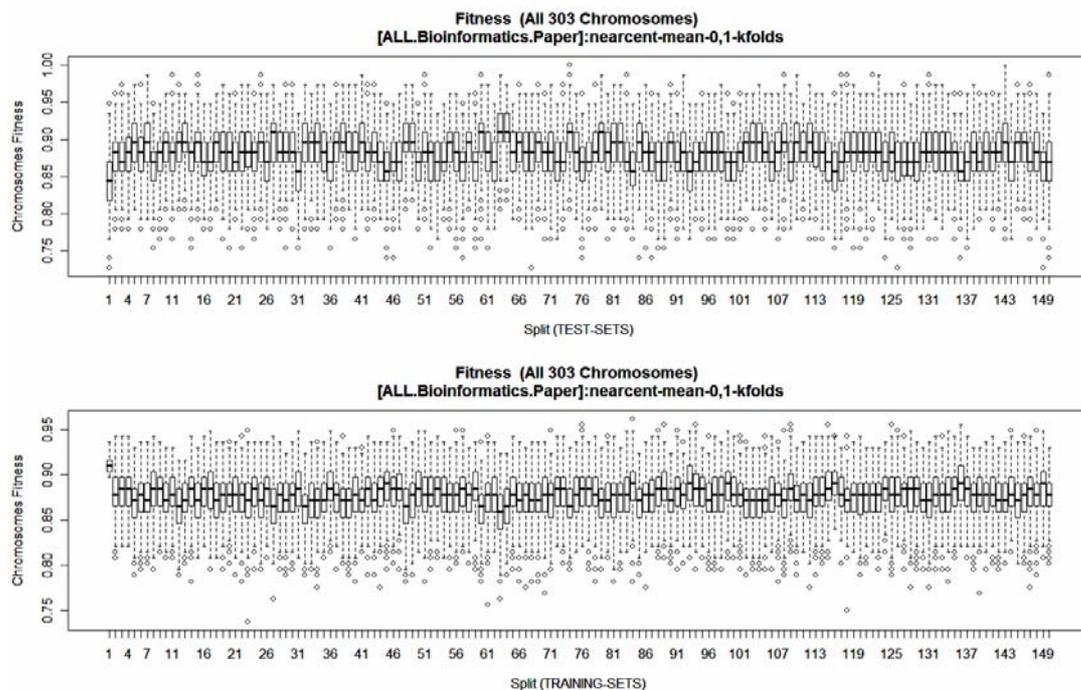
Figure 37 Chromosomes Fitness Distribution Across Splits. Top: in test sets. Bottom: in training sets.

Note the first box in the bottom plot in Figure 37 that shows the error in training set used by the GA. By definition, it must be "higher" than 0.9 due to restriction in fitness goal specified. Nevertheless, "outliers" dots below 0.9 in first box may appear showing those chromosomes that did not reach the fitness goal after the maximum number of generations.

Finally, using the following code, we can obtain the overall accuracy shown in Figure 38.

```
> plot(bb.nc, type="fitnesssplits")
```
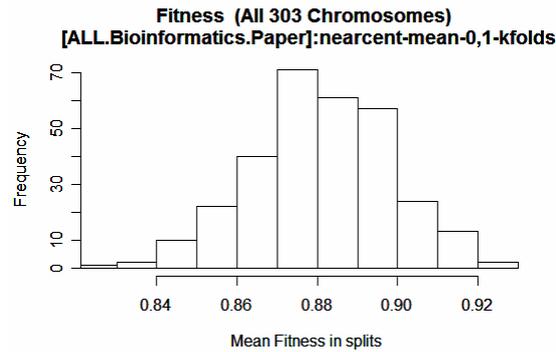
Figure 38 Overall Accuracy.

5.3.3 Gene Accuracies

To show the fitness of chromosomes for top-ranked genes, we can use the following code.

```
> plot(bb.nc,type="rankfitness")
```

Figure 39 shows the result. For example, for the first top-gene, the boxplot shows the fitness distribution of all models containing this first gene. Therefore, this plot might reveal relations between gene and fitness.
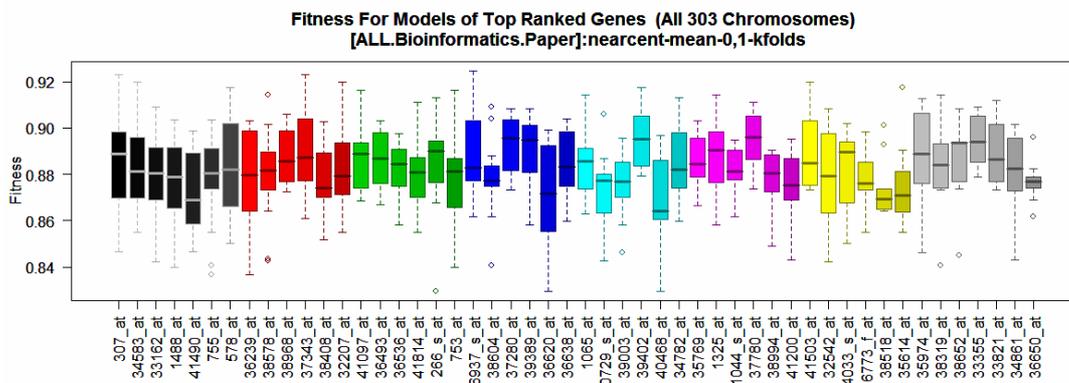

Figure 39 Fitness of chromosomes for top-ranked genes.

5.4 Model Refinement

As shown in section 2.4.4, one possible problem is that genes inside chromosome are not really needed. Therefore, sometimes it is advisable to remove these unnecessary genes. To assess the genes that are important in each model, we have implemented a classical backward selection strategy removing one gene at the time recursively (type ?geneBackwardElimination and ?robustGeneBackwardElimination). One can use this approach just after the GA threw the solution using the *callBackFuncBB* parameter in

configBB.VarSel, then saving automatically the shrunken chromosome. To do this, use something like the following code.

```
> bb <- configBB.VarSel(...,
    callEnhancerFunc=function(chr, parentBB) {
        robustGeneBackwardElimination(chr, parentBB,
            result="shortest")
})
```

Note, however, that backward elimination strategy is somehow inefficient when the starting chromosome size is large (10 or more).

To show the "optimal" gene size evaluated in the original split (assuming that no enhancement function has been provided), do the following.

```
> tchr <- lapply(bb.nc$bestChromosomes,
robustGeneBackwardElimination, bb.nc, result="shortest",
set=c(0,1), splits=1)
> barplot(table(unlist(lapply(tchr,length))),main="Length
of Shrunken Chromosomes\nEvaluated in Training Set in the
original split")
```
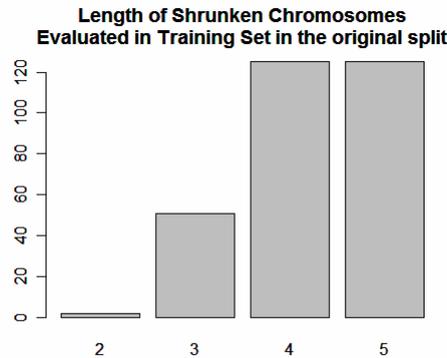

Figure 40 Shrunken chromosomes

Figure 40 shows that not all genes were necessary in more than a half of chromosomes evaluating in the training sets in only one split. We did the last computation for illustrative purposes. It would be better to assess the optimal chromosome size using the test error in all splits tough.

In general, this mechanism can be used to design a second level "search" or assessment of the evolved chromosomes.

5.5 Assessing GA and CPU Performance

The GA search (step 2) is, by far, the most time consuming part of the process. Many parameters affect the search (see Goldberg (Goldberg 1989)), but in the

end this is reflected by the evolution of fitness value. The evolution of the fitness can be shown using the following code.

```
> plot(bb.nc, type="fitness")
```
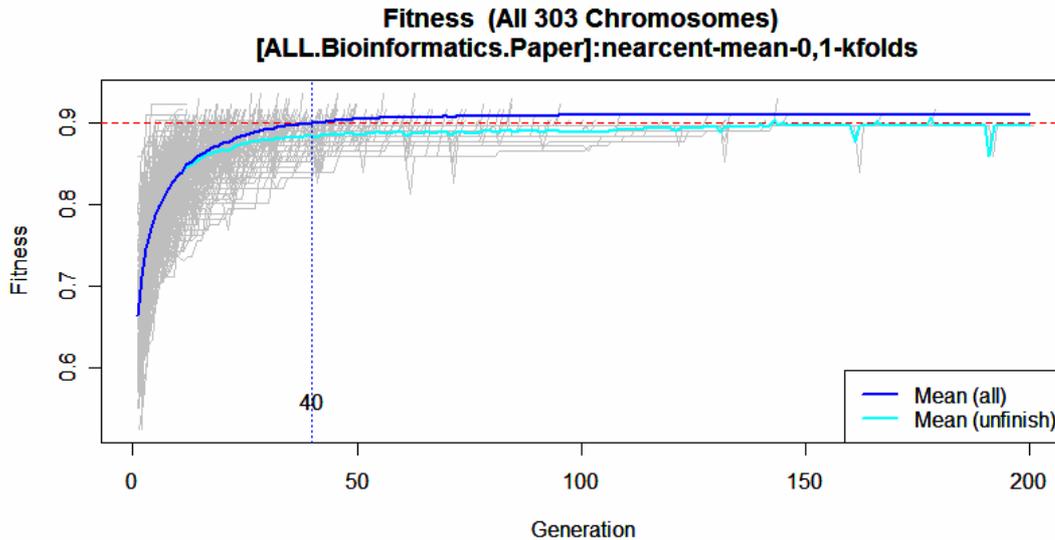

Figure 41 Evolution of the maximum fitness across generations in searches. The generation in which the average fitness reach the goal fitness (in red) is indicated (40). Mean fitness is the fitness considering all searches and represent the expected frequency by generation. "Unfinish" fitness average is the fitness of all searches that has not ended by a given generation, it intend to show the average worst case expectation.

Figure 41 shows the fitness evolution across generations, which indicate that the goal fitness is reach, in average, in 40 generations. This plot can help in evaluating and comparing different search configurations. We can not see in this figure, however, what is the evolution of the attainable fitness because many searches finish after reaching the goal fitness 0.9. One way to see the maximum attainable fitness under certain configuration is setting and "unreachable" fitness (see section 8.3).

A similar plot, which is used in the default monitoring system, can be shown using the following code (Figure 42).

```
> plot(bb.nc, type="generation")
```

To assess the time spent, we can use the "timing" variable stored in the BigBang object, as follows.

```
> plot(bb.nc$timing)
> plot(bb.nc$timing,bb.nc$generation)
```
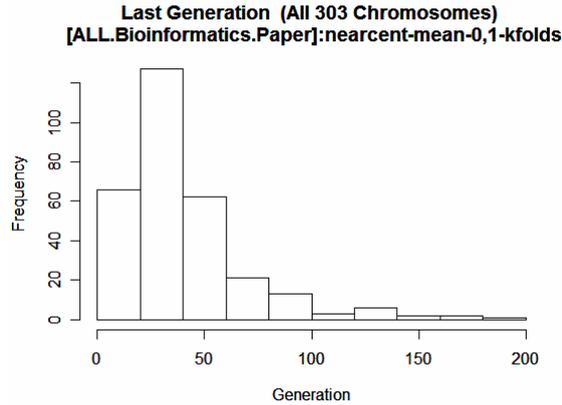
**Figure 42 Distribution of the last generation.**
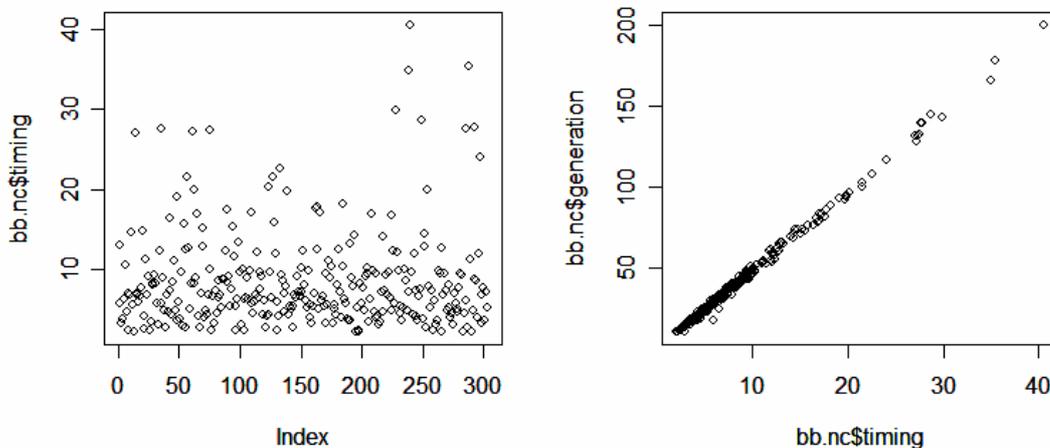
Figure 43 Performance of the search. Left: seconds spend by each search. Right: seconds per generation, which is given by the slope.

5.6 Chromosome Visualization

There are 6 common plots to visualize the models. In this section, we will describe briefly each visual representation along with code examples.

5.6.1 Heatmaps

Heatmap is the one of most common visual representation of gene expression in across samples. The method *heatmapModels* (?heatmapModels.BigBang) uses the common R *heatmap* function inheriting its parameters (?heatmap). In a heatmap, the expression values are commonly normalized (*scale* parameter) and converted to colour scales. In general, better visualization is obtained when a hierarchical clustering is performed in both, genes in rows, and samples in columns. However, because the unsupervised hierarchical clustering methods may cluster samples differently than the classification method of our choice, it could, instead, mislead the user (*ColV* and *RowV* parameters). Gene names, sample names, colours, ordering scheme

(hierarchical clustering), and scaling can be controlled by parameters. For an example, use the following code (Figure 44).

```
> heatmapModels(bb.nc,bb.nc$bestChromosomes[[12]])
> heatmapModels(bb.nc,bb.nc$bestChromosomes[[12]]),col=-
5, Colv=NA, Rowv=NA)
```
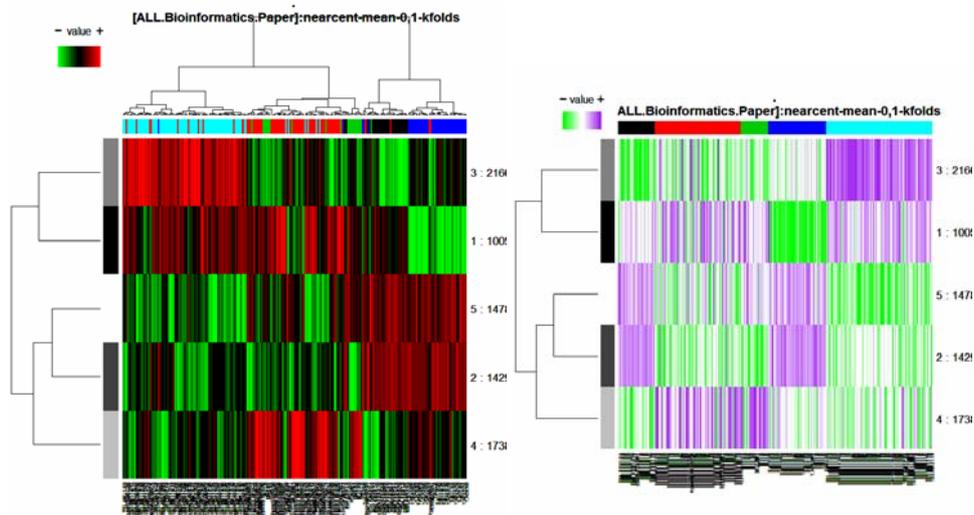

Figure 44 Heatmap representations of a model. Genes in vertical, samples in horizontal. Both may or may not have an ordering scheme (e.g. clustering). Profile colour can be customized easily. Sample colour (top) is class dependent. Gene colours (left "greyed" bars) indicate index in the original chromosome.

5.6.2 Principal Component Space

Another useful representation is the principal component space (PCA). In PCA each component is a linear combination of the original gene values, in a way that each component is uncorrelated each other (?pcaModels.BigBang and ?prcomp). The final components are sorted by the amount the original variance that each component explains from the original data variation. Therefore, PCA is seen in statistics as a data dimension reduction technique. When similar samples share similar gene profiles, in the PCA space they may have similar components. Thus, if gene profiles have been already selected for its ability to classify samples it would be likely that samples of same class would appear closer each other in a PCA plot. For plotting the PCA use the following code (Figure 45).

```
> pcaModels(bb.nc,bb.nc$bestChromosomes[[1]])
```
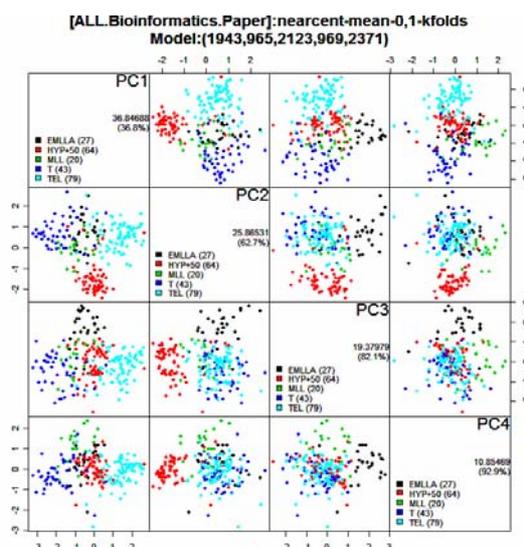
Figure 45 Principal Component Space of profiles of a model. In first row, the first principal component is plotted vertically and the other components horizontally. Similarly, in the first column, the first principal component is drawn in horizontal. The percentage of the variance explained by each component and the accumulated variance is shown. Colours indicate class.

5.6.3 Raw Values

Sometimes we would like to see a representation of the original values without any transformation at all. Figure 47Figure 46 shows the output of the code below. Note that samples are sorted by class but the relative position is arbitrary.

```
> plot(bb.nc, bb.nc$bestChromosomes[[1]],
type="genevalues")
```
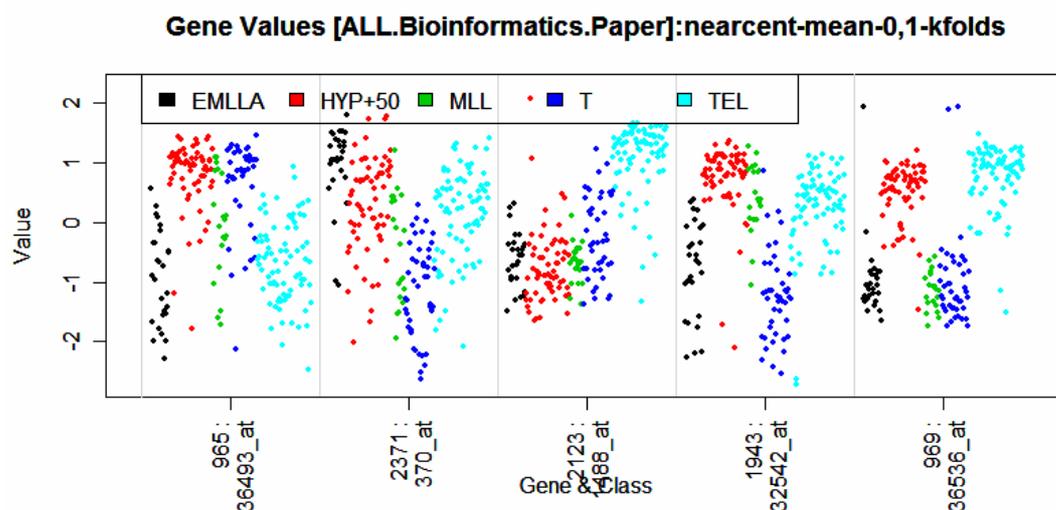

Figure 46 Raw Values of a model per gene. Raw value in vertical. Gene in horizontal (false relative sample position).

5.6.4 Distribution of Raw Values

Another way to represent the raw values is comparing the distribution of raw values.

```
> plot(bb.nc, bb.nc$bestChromosomes[[1]],
type="genevaluesbox")
```
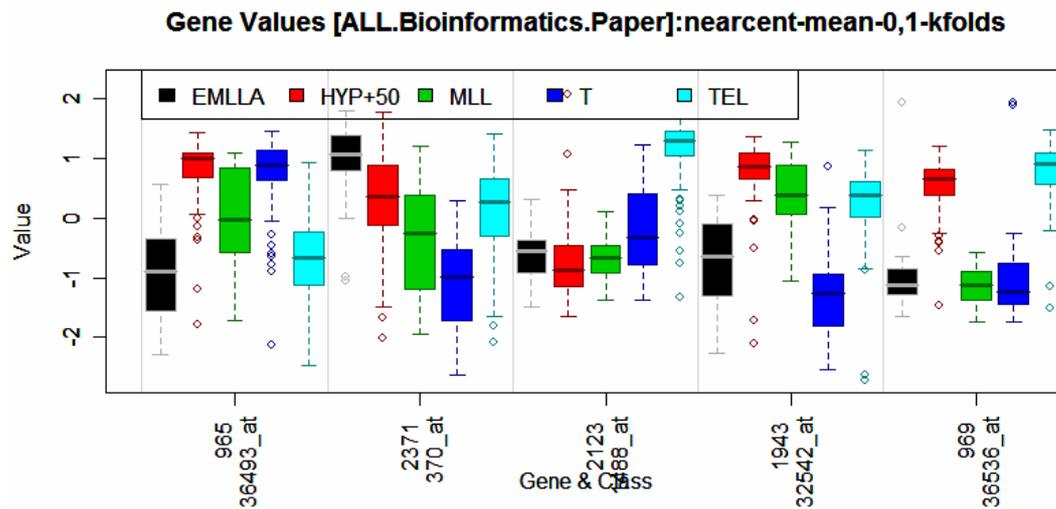

Figure 47 Distribution of raw values per class across genes.

5.6.5 Gene Profiles per Class

```
> plot(bb.nc, bb.nc$bestChromosomes[[1]],
type="geneprofiles")
```
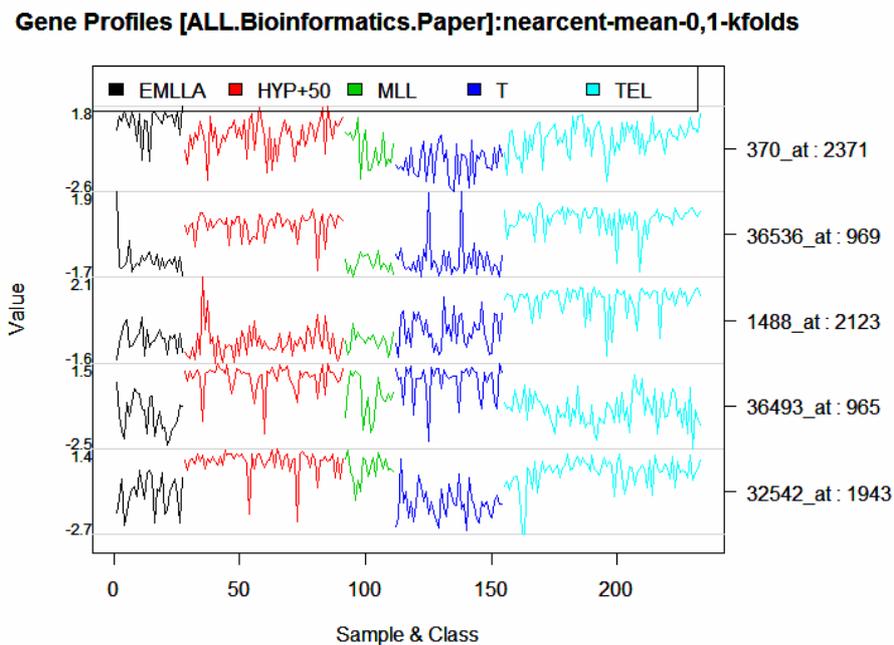

Figure 48 Gene Profile per Class.

5.6.6 Sample Profiles per Class

```
> plot(bb.nc, bb.nc$bestChromosomes[[1]],
type="sampleprofiles")
```
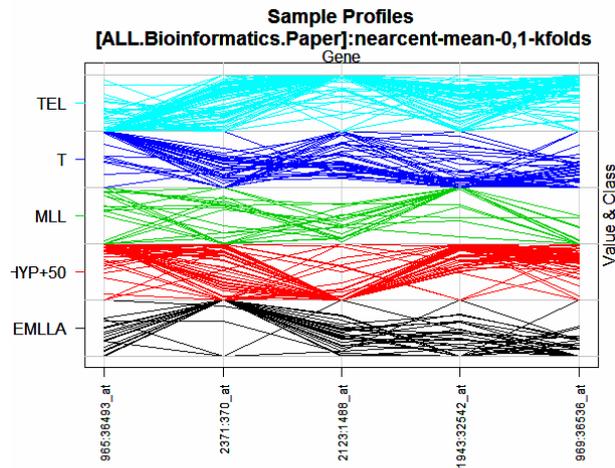

Figure 49 Sample Profiles. Sample expressions are scaled to range 0 to 1.

5.7 Predicting Class Membership of Unknown Samples

One of the goals of the analysis is the prediction of unknown samples, or samples that were, in purpose, left out of the analysis. For this, we can use the parameters *force.train* and *force.test* of *configBB.VarSel* prior the analysis. Other strategy could be just adding the new data subsequent to the analysis. In this case we could use the *predict* method. As an illustrative example, we will use the first 10 original samples as "new unknown" samples in the following code.

```
> new.cm <- predict(bb.nc, t(bb.nc$data$data[1:10,]),
newClass="NEW", permanent=TRUE)
> new.cm
> plot(bb.nc, type="confusionbox")
```

predict method adds the new data temporary to the data variable (bb.nc$data$data), assigns new classes, and executes *classPredictionMatrix*. If *permanent* is true, the data will remain there after *predict* finish. As expected, the 10 new samples added are predicted as EMLLA (Figure 50).
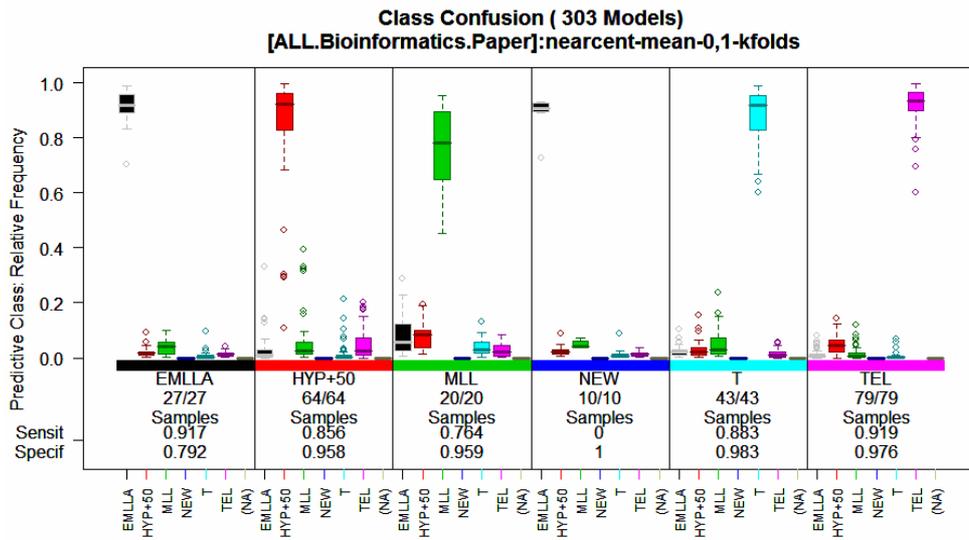
Figure 50 Prediction of "NEW" samples as EMLLA (label "NEW").

6   Step 4 - Developing Representative Models

The result of the evolutionary process is a large list of chromosomes, and we have assumed that the frequency of the genes in these solutions is linked to their importance regarding the classification problem. We need then, a method to create a model that select from this list of most important genes, the ones that maximize classification accuracy. A simple method is following the classical forward selection method, adding one gene at the time starting from the most frequent to the least frequent. To obtain the best models using this method, use the following code.

```
> fsm <- forwardSelectionModels(bb.nc)
> fsm$models
```
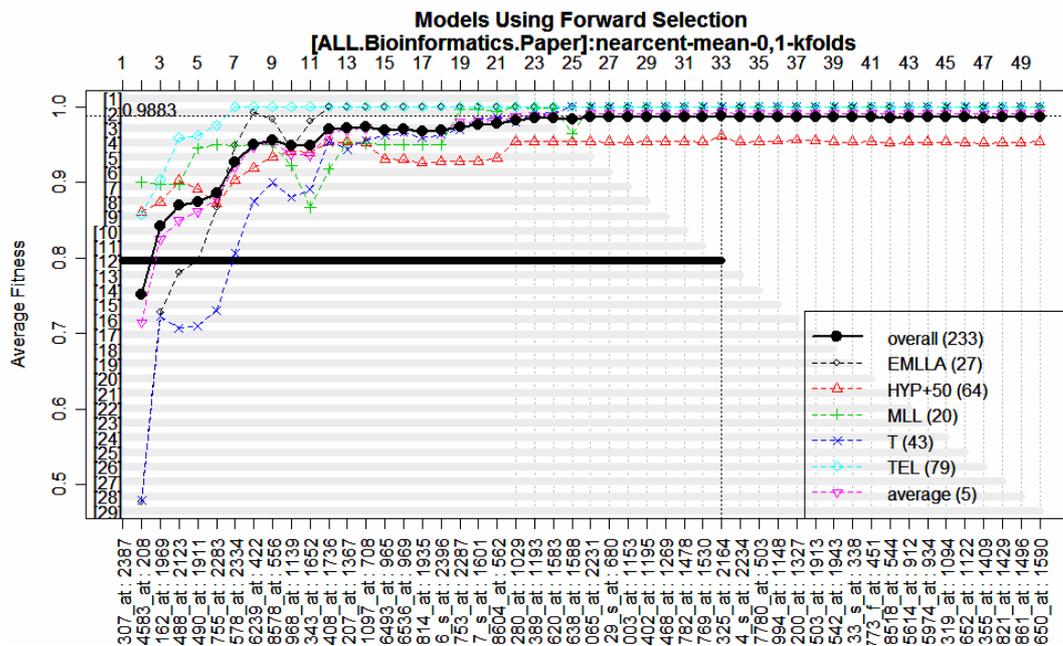

Figure 51 Representative Model by Forward Selection using the most frequent genes. Solid line represents the overall accuracy (misclassified samples divided by the total number of samples). Coloured dashed lines represent the accuracy per class. Average represents the average class accuracy (the sum of averages per class divided by the number of classes). 29 models resulted from the selection whose fitness value is 0.99 times the maximum. The best model is 12, which is formed from the 1st gene up to the 33rd gene.

```
> length(fm$models[[12]])
# ~33
```

Confusion Matrix for the designed model:
```
> confusionMatrix(bb.nc, chromosomes=fm$models[[12]])
```

Confusion Matrix for the top-10 genes:

```
> confusionMatrix(bb.nc,
chromosomes=fm$models[[12]][1:10]) # first ten genes
```

Confusion Matrix for the first evolved chromosome, evaluated in the first training set (first split):

```
> confusionMatrix(bb.nc,
chromosomes=bb.nc$bestChromosomes[[1]], set=c(1,0),
split=1)
```

6.1 Plotting Representative Models

```
> plot(bb.nc, chromosomes=list(fm$models[[12]]),
type="confusion")
```



```
> heatmapModels(bb.nc, fm, subset=12)
```
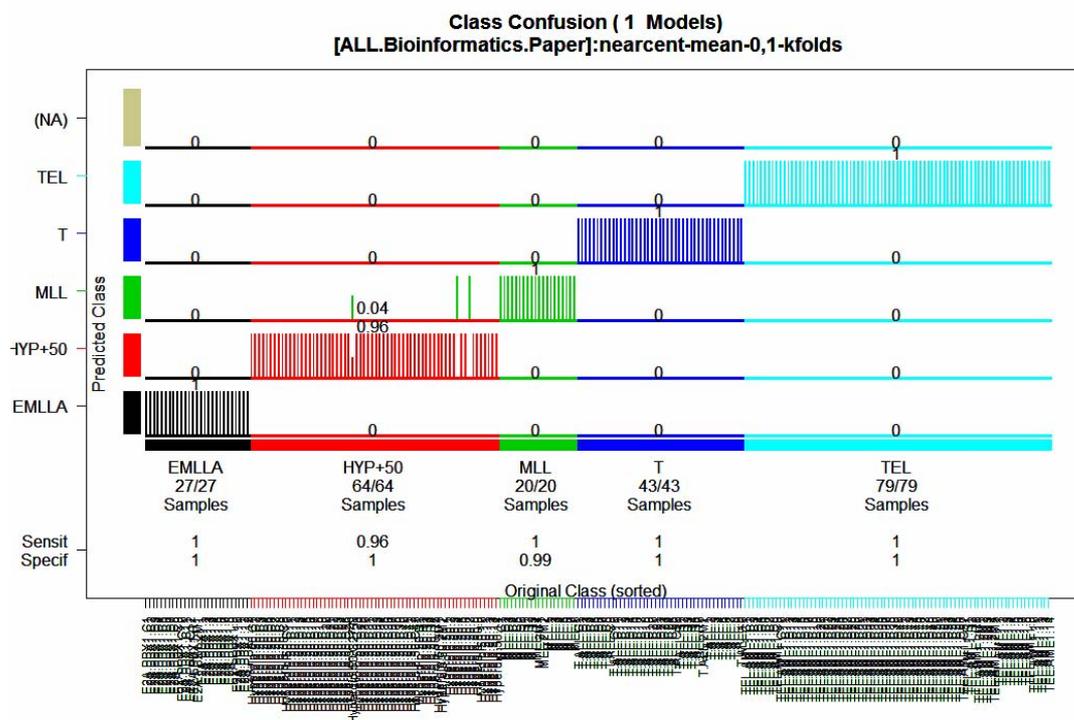
[ALL.Bioinformatics.Paper]:nearcent-mean-0,1-kfolds
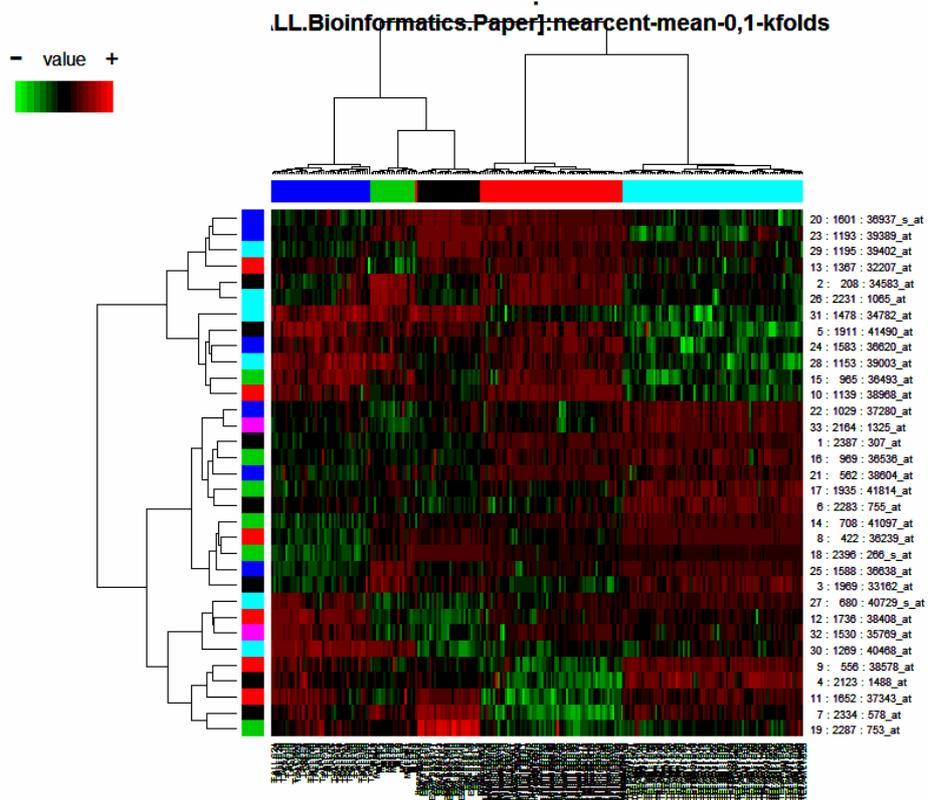
```
> pcaModels(bb.nc, fm, subset=12)
```


[ALL.Bioinformatics.Paper]:nearcent-mean-0,1-kfolds
56,1139,1652,1736,1367,708,965,969,1935,2396,2287,1601,562,1029,1193,158
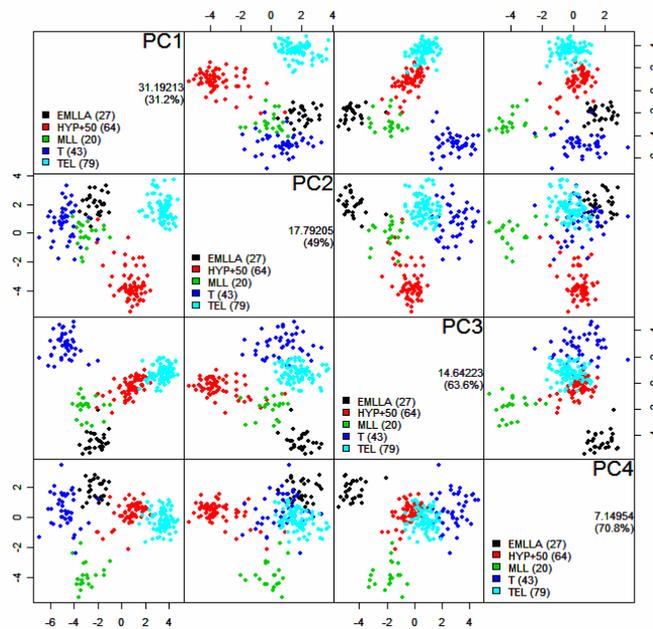
```
> plot(bb.nc,fm$models[[12]],type="genevalues")
```
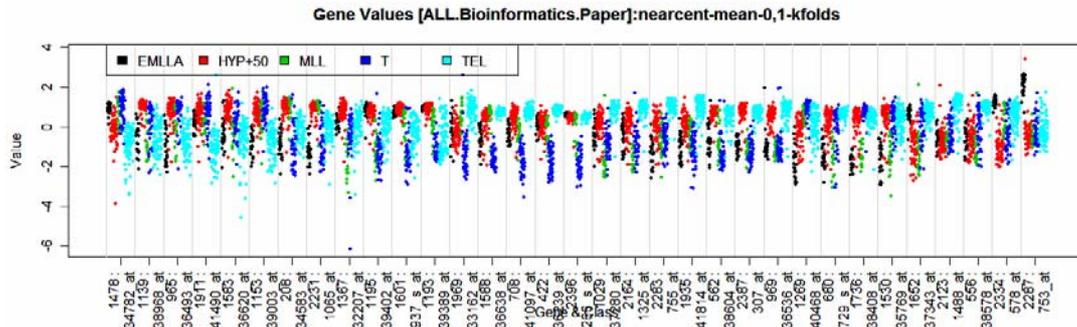
```
> plot(bb.nc,fm$models[[12]],type="genevaluesbox")
```
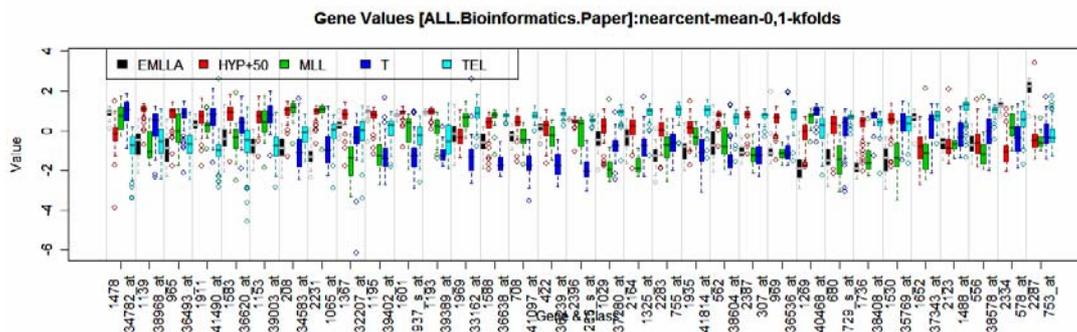


```
> plot(bb.nc,fm$models[[12]],type="geneprofiles")
```
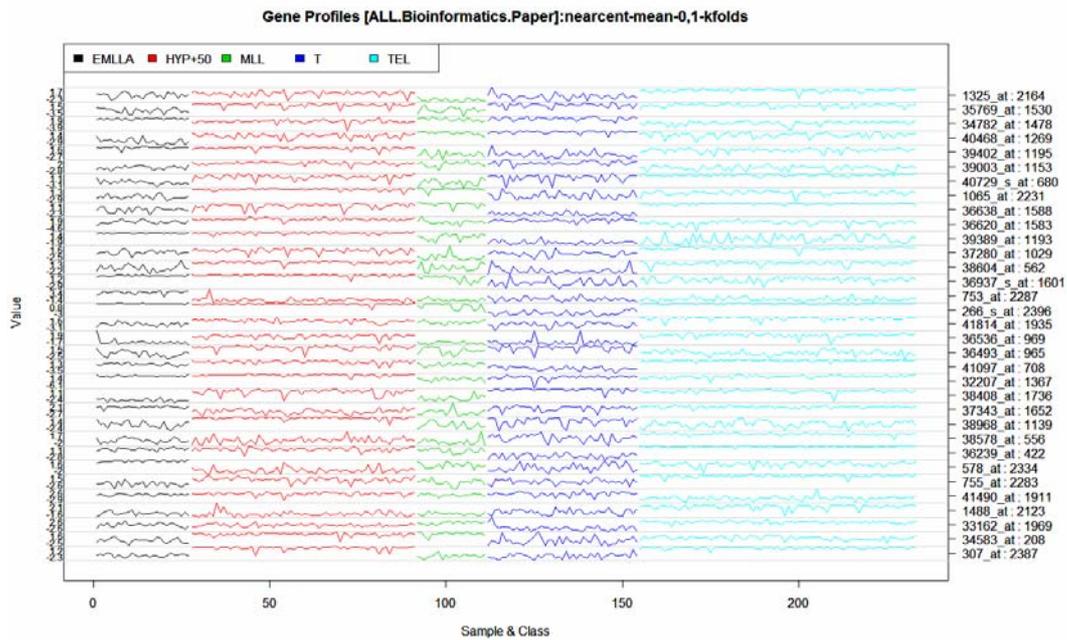


```
> plot(bb.nc,fm$models[[12]],type="sampleprofiles")
```
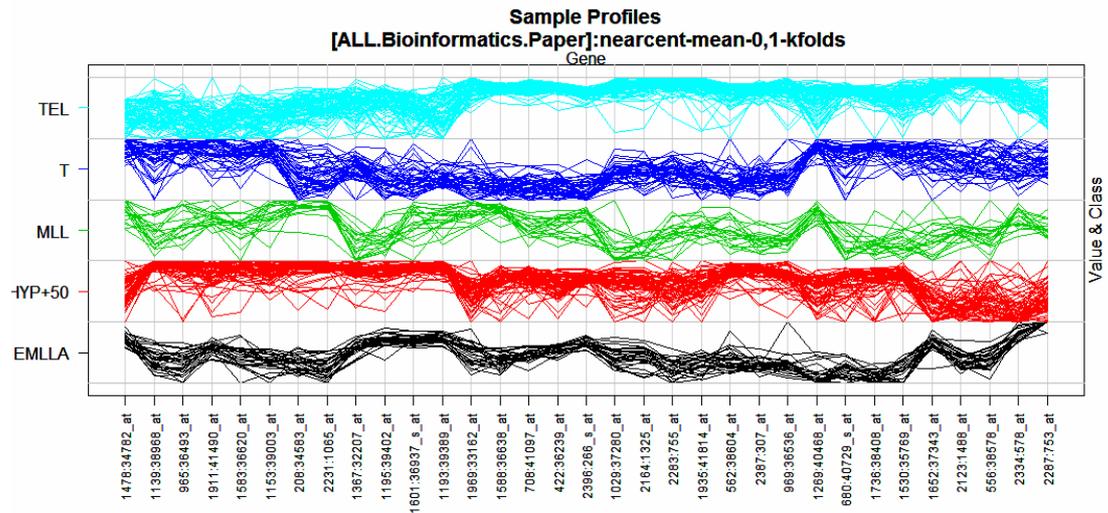
6.2 Predicting Unknown Samples

As in section 5.7, for an illustrative example for prediction, we will use the first 10 original samples as "new unknown" samples in the following code (see the column "NEW" in Figure 52).

```
> new.cm <- predict(bb.nc, t(bb.nc$data$data[1:10,]),
newClass="NEW", permanent=TRUE, func=NULL)
> plot(bb.nc, chromosomes=list(fm$models[[12]]),
type="confusion")
```
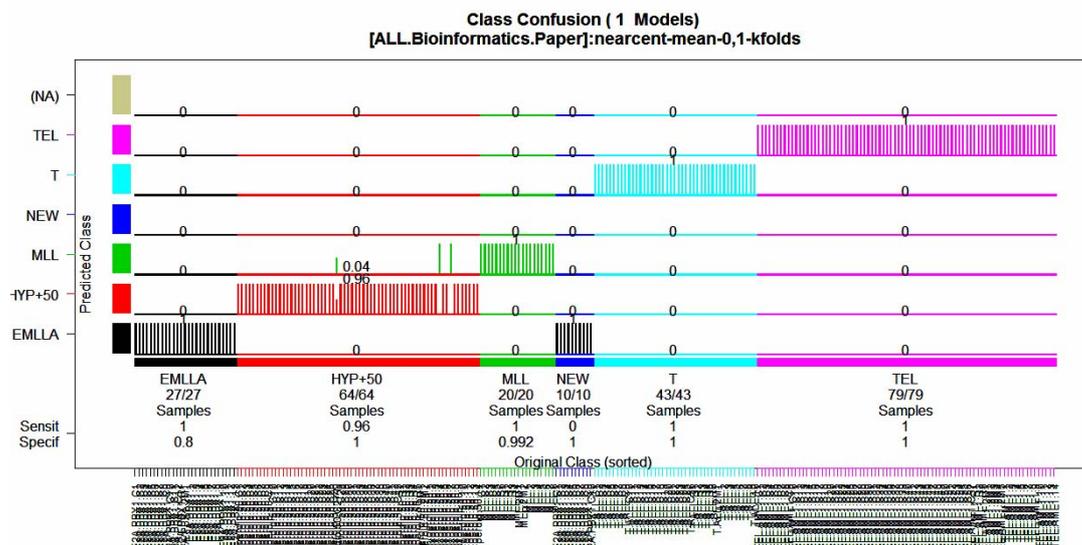

Figure 52 Prediction of "NEW" samples.

7 Additional Options

7.1 Regression and Survival Analysis

A benefit of GALGO is that it can be easily expanded to solve other kind of problems. For instance, the implementation of a fitness function for a regression problem could be done as following.

```
reg.fitness <- function(chr, parent,tr,te,res) {
   try <-  parent$data$dependent[tr]
   trd <-
data.frame(parent$data$data[tr,as.numeric(chr)])
   colnames(trd) <- c("g1","g2","g3")
   trm <- lm(try ~ g1+g2+g3+g1:g2+g1:g3+g2:g3, data=trd)
   tey <- parent$data$dependent[te]
   ted <-
data.frame(parent$data$data[te,as.numeric(chr)])
   colnames(ted) <- c("g1","g2","g3")
   cor(predict(trm,newdata=ted),tey)^2
}

#####
# This code try to regress the values
# of a random selected gene.
#####
# "userFile.txt" contains ALL and ALL.classes,
# otherwise we can use data=ALL and strata=ALL.classes
reg.bb <- configBB.VarSelMisc (file="userFile.txt",
chromosomeSize=3, niches=1, maxSolutions=1000,
goalFitness = 0.5, saveVariable="reg.bb",
saveFrequency=50, saveFile="reg.bb.Rdata",
fitnessFunc=reg.fitness)

#Now, choose the gene
regI <- sample(1:ncol(reg.bb$data$data), 1)
#Set the values of the dependent variable,
#essential to reg.fitness
reg.bb$data$dependent <- reg.bb$data$data[,regI]
#set to zero to avoid selection of that gene
reg.bb$data$data[,regI] <- 0

blast(reg.bb)
```

In the same manner, survival analysis can be implemented providing only the fitness function. Note that the example shown here considers always three genes, if a forward selection is used, even that the chromosome contains more

than three genes, only the first three genes would be used. The user has to generalize the function considering all the genes present in the chromosome.

7.2 *Parallelization*

In this first release, we have implemented a semi-automatic and very simplistic form of parallelization benefiting from the file saving scheme. The core idea is depicted in Figure 53, where autonomous processes save their results progressively in independent files that can be merged by the analysis process.
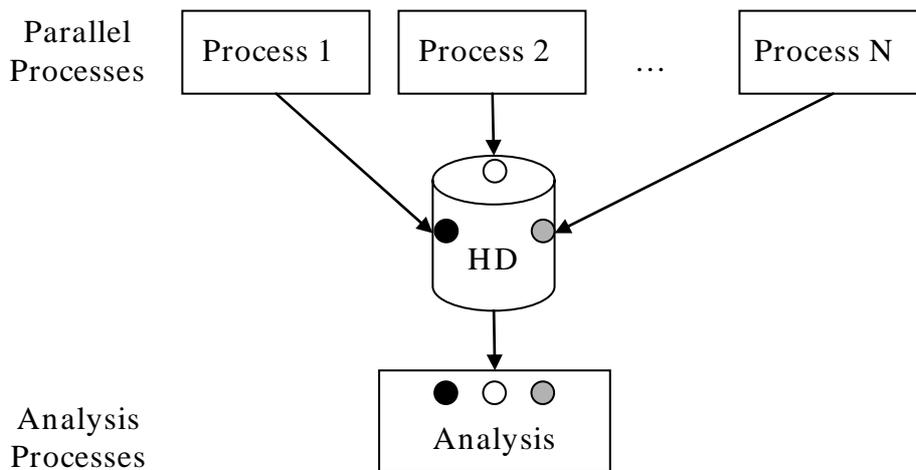

Figure 53 Parallelization of GALGO processes. Many parallel process running independently (on different PCs or CPUs) and save their results in separate files on a common storage device (e.g. Hard Disk). These files are then merged in a BigBang Object and the analysis is performed as described in this tutorial.

The following code can be used to parallelize a given task.

```
#Initial process:
#load data and configure initial objects run once
library(galgo)
bb <- configBB.varSel(..., saveFile="bb.parallel.Rdata",
...)
saveObject(bb)
#

#Parallel process:
#run as many process you want
library(galgo)
loadObject("bb.parallel.Rdata")
assignParallelFile(bb)
blast(bb)
#
```

```
#Analysis Process:
library(galgo)
loadObject("bb.parallel.Rdata")
loadParallelFiles(bb)
plot(bb) #further analysis…
#
```

In the above code, the "Initial process" section serves to create the master object, which will be "copied" in all parallel process. The section "Parallel process" is the code used in every parallel process, which assign a unique consecutive file for the process in turn. The "blast" method save the results in these files progressively. Finally, in the "Analysis Process" section, which must be another independent process, the master object is loaded and the results from the updated files are merged to perform the analysis.

7.3 Using weighted variables

It may be desirable to subset variables on the basis of a defined property and explore preferentially certain subsets. This can be done in GALGO using a weighting scheme that gives different weights to variables belonging to a certain category. Technically, this can be performed specifying a new random generation function for the Gene object as follows.

```
geneWeights <- ...
# a vector representing the probability of each variable
to be included in a chromosome.

runif.weigths <- function(.O, n, mn, mx) {
     sample(mn:mx, size=n, prob=geneWeights)
}
bb <- configBB.VarSel/Misc(..., geneFunc=runif.weights)
```

7.4 Using GALGO to solve a classical Optimization Problem

A classical example in genetic algorithms is a regression problem with binary representation. It has several limitations but is shown here for illustrative purposes. Let assume that we have a series of samples changing in time, and that we want to compute the slope and intercept of a linear model. We will approach this creating a binary chromosome of size 10 which would be interpreted as an integer ranging from zero to 2^5-1 for both, the slope and the intercept. Now, we need a stopping rule, a condition that assures that the current approximation is good enough, or in the other hand, that the evolution do not last forever.

The first task is to solve the gene representation, which is binary, only 0 or 1 can be represented in a gene. Thus, we create the Gene.

```
ge <- Gene(0, 0, 1)  # 0 - id, 0 - minimum, 1 - maximum
ge
```

The second task is to create a chromosome that group five genes of this type and a decode function that converts the binary representation into real values.

```
decodeChr <- function(cr) {
    chr <- as.numeric(cr)
    c(a= sum(2^(4:0) * chr[1:5]), b= sum(2^(4:0) *
    chr[6:10]))
}
cr <- Chromosome(genes = newCollection(ge, 10),
decode=decodeChr)
cr
```

Now we use this chromosome prototype to create our population of chromosomes that is necessary for the Galgo object.

```
ni <- Niche(chromosomes = newRandomCollection(cr, 10),
elitism=1)
ni
```

Before creating the galgo object, we need the fitness function. It should receive the chromosome and a parent object, which may contain any data.

```
regFitnessFunc <- function(cr, parent) {
    chr <- decode(cr) # this really calls decodeChr
    1-sum((chr["a"] + chr["b"] * parent$x -
parent$y)^2)/(sum(parent$y^2)-
sum(parent$y)^2/length(parent$y))
}
```

Then, we create the Galgo object.

```
ga <- Galgo(populations = ni, fitnessFunc =
regFitnessFunc, goalFitness = 0.88, maxGenerations = 50,
callBackFunc=plot)
```

Now we need to create random data to run the test.

```
data <- list()
data$x <- 1:50
a <- runif(1,20,31)
b <- runif(1,4,16)
```

```
data$y <- a + b * data$x + rnorm(50, m=0, sd=100)
```

Finally, we run the evolution and compare the best solution.

```
evolve(ga, parent=data)
a
b
decodeChr(best(ga))
```

```
data$y <- a + b * data$x + rnorm(50, m=0, sd=100)
```

8 Parameter Estimation and Considerations

8.1 Number of Solutions

In the analysis of the population of selected chromosomes, it is important to rely on a sufficiently large number of chromosomes. The ability to observe in real time the stability of the gene composition in the chromosome population is an important tool to select an appropriate number of chromosomes. In practice, for many applications, the number of solutions, required to stabilize the top 20 most frequent genes is sufficient. From Figure 26, we can estimate that between 1000 and 2000 solutions are needed to stabilize 20 genes for this dataset. If more genes are strictly needed, a longer run would be required. For example, Figure 54 shows that 8000 solutions would be needed to stabilize the first 50 genes (look around -15000 in vertical axis). However, exact positions within these 50 are eventually changing. A broader plot shown in Figure 55 reveals interesting results, the first ~300 genes (in black) has been relatively stable along the run whereas the next ~300 genes (in red) has been increasingly stabilizing.
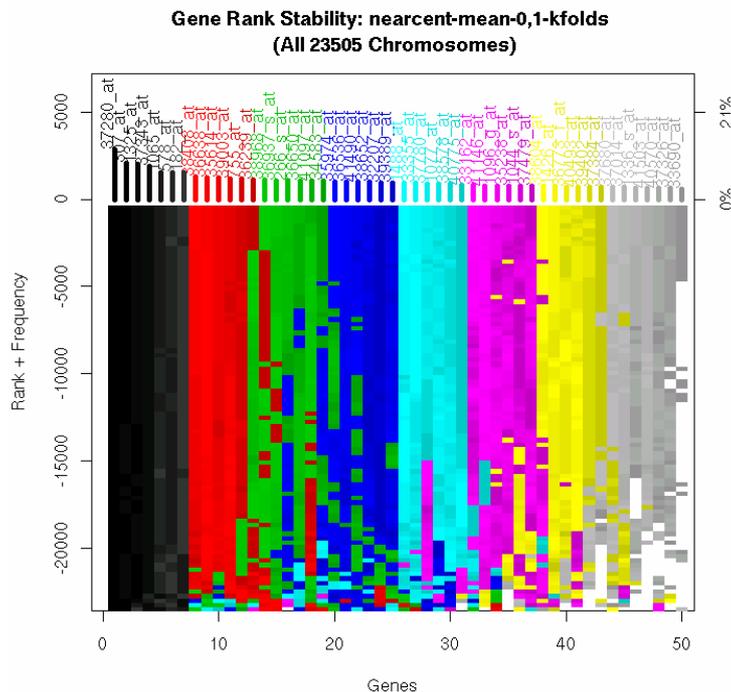

Figure 54 Gene Rank Stability for the first 50 genes in a long run (23505 solutions).

Another important issue related to the number of solutions is about reproducibility. The frequency computed in two replicated experiments should be approximately the same when the rank is not due to random fluctuations. Therefore, a scatter plot of gene frequency from two replicated

searches, and the number of overlapped genes would reveal stability between runs. Figure 56 show that as more solutions as more stable the frequency, and more top-overlapped genes between independent runs.
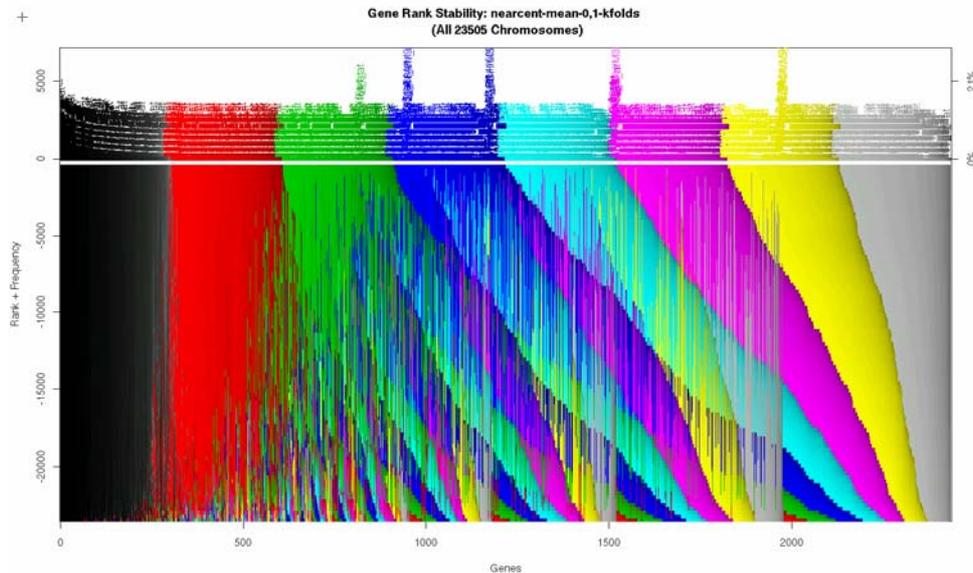

Figure 55 Gene Rank Stability for all genes in a long run (23505 solutions).
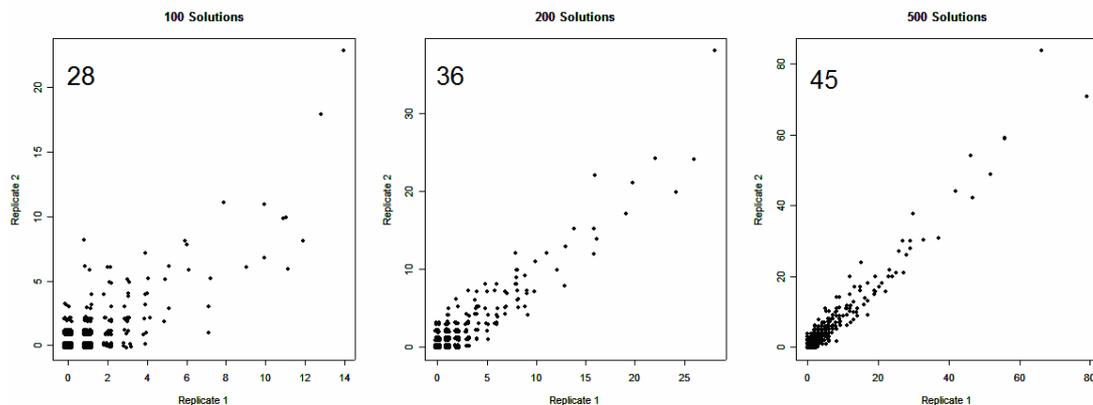

Figure 56 Gene Frequency Comparisons. Comparisons were made in replicates at different number of solutions. Number inset represent the number of overlapped genes in the top 50.

8.2 Number of Generations

Ooi *et al.* (Ooi *et al.* 2003) have used a 100 as the number of generations whereas other researchers (Liu *et al.* 2005; Peng *et al.* 2003) have used instead, a large number of generations (from 2,000 to 100,000). Nevertheless, commonly the fitness grows exponentially until a plateau effect where the fitness grows very slowly or stops its growth (Figure 9). Therefore, we consider pointless to use thousands of generations. In such a case, we think it would end up in overfitting. To investigate the maximum possible fitness and

then decide the number of generations, we could execute a preliminary run using an unreachable fitness in 20 or 50 cycles as shown in Figure 57 (see section 8.3).
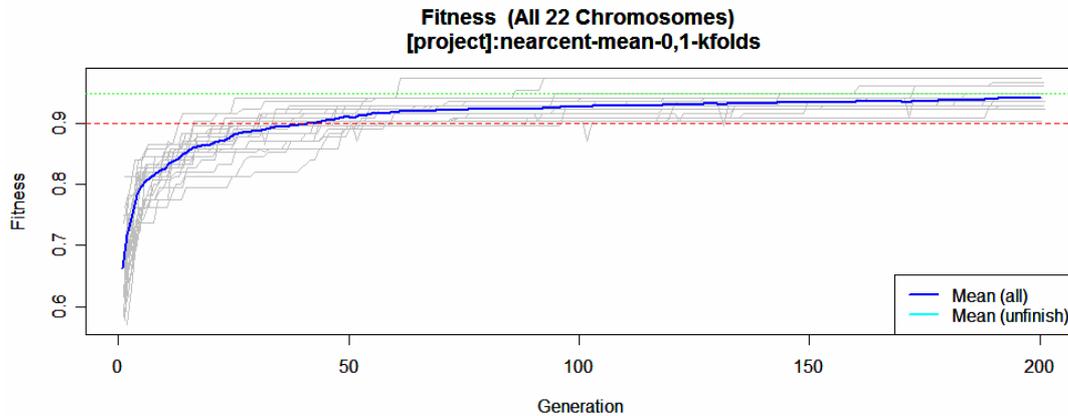

Figure 57 Fitness evolutions in 20 solutions using an unreachable fitness.

8.3 Goal Fitness

The goal fitness should be the average reachable fitness in a reasonable amount of generations (see previous section). Commonly, we choose the average fitness resulted after 30, 50, or 100 generations, but it is problem dependent. Perhaps the plot type "fitness" using between 20 and 50 solutions would help in deciding the goal fitness to use (Figure 57). Therefore, a short run with a very high goal fitness value (to see the plateau effect) can be useful to make a decision in a goal fitness, which can be done using the following.

```
> data(ALL)
> data(ALL.classes)
> bb <- configBB.VarSel(data=ALL, classes=ALL.classes,
chromosomeSize=5, classification.method="nearcent",
maxSolutions=20, goalFitness = 1, maxGenerations=200,
maxBigBangs=20)
> blast(bb)
> mf <- meanFitness(bb)
> mf[c(25,50,75,100,200)]
```

You must consider that the final fitness based on model selection methods (as those from forward selection method using top ranked genes) is commonly higher that those from evolutionary process. We have seen in several datasets that the fitness computed by forward selection method after pooling many solutions, is between 5% and 15% higher that the attainable goal fitness used for evolutionary solutions.

8.4 Chromosome Size

The chromosome size has a great impact in the fitness and the performance. The higher the chromosomes size the slower the computation. Moreover, increasing the chromosome size not always generate average increase in fitness. Some researchers have used a high chromosome size (Li *et al.* 2001), nevertheless, in our research, the increase in size lead in serious overfitting problems, as shown in Figure 58. Certainly, the fitness is slightly higher for longer chromosomes when evaluated in the split used in the selection of the genetic algorithm. However, when the same chromosomes are evaluated in all splits, the average fitness is drastically decreased. This means that for longer chromosomes, the fitness is more dependent on the specific split used in the search, hence overfitted.
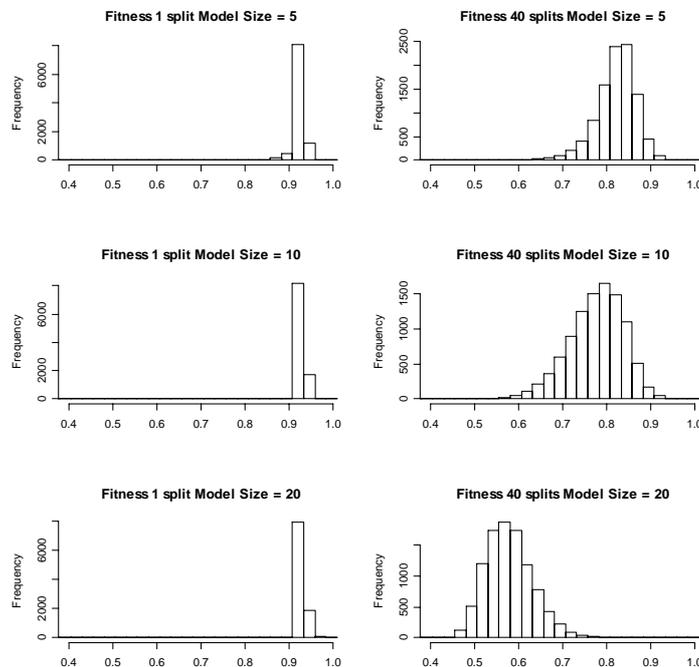

Figure 58 Fitness distribution using different model sizes (chromosome size). Left panels correspond to fitness evaluated in one split only, the one used for the evolution in the genetic algorithm. Right panels correspond to the fitness evaluated in all splits. Data consist on 40 prostate cancer samples (Lapointe *et al.* 2004).

On the other hand, smaller chromosome sizes could not reach appropriate goal fitness, or they may be not robust enough. For classification problems, we have seen that, in general, the number of different classes gives a lower limit for the number of genes in a chromosome. It makes sense because genes tend to be unimodal among samples (but not unique to a class). Chromosome sizes smaller than three may be meaningless because an exhaustive search might be faster and global than those searches using any stochastic method.

8.5 Population Size

Increasing the population size would explore more solutions-landscape per generation, thus a reduction in the number of the generations is expected. However, larger populations would require more process time. Hence, the population size, the number of variables, and the number of generations are linked. The default population size in *configBB* functions is 20 plus an additional unit per each 400 variables, thus for 2,000 genes the population size would be 25, and for 10,000 genes it would be 40. Increasing the number of genes while keeping constant the population size would increase slightly the number of generations required. However, for more than 10,000 variables it may be worth to increase the population size manually. It is recommendable filtering the data eliminating variables with very small variance to avoid large processing times.

8.6 Elitism

In genetic algorithms, the new generation is randomly build based on the value of the fitness for every chromosome in the current generation. Because of the randomness of this process, the results are that the maximum fitness of the current generation fluctuates. The elitism mechanism ensures the conservation of the best chromosome in the next generation, which generally improves the performance. However, because the best chromosome is always present in the population, the system could be trapped in local peaks (attractors). Therefore, the default behaviour using *configBB* functions is to use elitism 95% of the time (in a cyclic fashion, nine generations with elitism turned on and one generation with a probability of elitism to 0.5). If poor fitness values are always obtained, the GA search could be early trapped in poor large attractors, decreasing the elitism (even to 0) many of times would solve the problem.

8.7 Number of Niches and Migration

A niche is a population that evolves by their own; it considers only the chromosomes in the current population for building the next generation. The default number of niches in *configBB* functions is one. The idea of isolating chromosomes into niches is to exchange individuals hoping that combinations may result in improvements. Without any exchange between niches, they would evolve independently in parallel. In this case, it would be a loss of processing time because the result of the evolution is only one chromosome, the best ever visited. Therefore, if the number of niches is higher than one, some kind of exchange must be established. The default behaviour is to migrate the best chromosome to other niches 7.5% of the time (in a cyclic

fashion, 18 generations with migrate off to set up good chromosomes, 1 generation with a probability of 0.5, and 1 generation with migration on). Nevertheless, this behaviour is only visible when the number of niches has been explicitly increased rather than using the default.

8.8   Mutations and Crossover

In genetic algorithms, generally the mutations and crossover are set up by probabilities. A 0.01 probability of mutation would mutate in average 1 of 100 genes, or 1 mutation every 20 chromosomes of size 5. In the literature, common values for mutation are even smaller. Nevertheless, those values are effective for binary encoded chromosomes, where the number of "genes" for a single parameter could be very large. Therefore, for variable selection in microarray data we have to adjust this probability. In this package, we use direct integer representations in *configBB* functions for convenience, and the mutation probability is by far higher than in the binary-based genetic algorithms. Hence, the default has been set to in average, one mutation per chromosome (actually, the number of mutations in the populations is the same than the number of chromosomes). For crossover, the default value involves all chromosomes in the exchange. Alternatively, GALGO can interpret mutation and crossover probabilities or mutation and crossover absolute numbers (type ?Niche in R). We have used numbers instead, because it is easier for us to think on mutations per chromosome or per population than mutations per gene.

REFERENCES

Efron, B., and Tibshirani, R. J. (1993). An introduction to the bootstrap. New York: Chapman & Hall.

Goldberg, D. E. (1989). <u>Genetic algorithms in search, optimization, and machine learning</u>. Reading, Mass., Addison-Wesley Pub. Co.

Holland, J. H. (1975). <u>Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence</u>. Ann Arbor, University of Michigan Press.

Lapointe, J., Li, C., Higgins, J. P., van de Rijn, M., Bair, E., Montgomery, K., Ferrari, M., Egevad, L., Rayford, W., Bergerheim, U., Ekman, P., DeMarzo, A. M., Tibshirani, R., Botstein, D., Brown, P. O., Brooks, J. D. and Pollack, J. R. (2004). "Gene expression profiling identifies clinically relevant subtypes of prostate cancer." <u>Proceedings of the National Academy of Sciences of the United States of America</u> **101**(3): 811-816.

Lattin, J. M., Carroll, J. D., Green, P. E. and Green, P. E. (2003). <u>Analyzing multivariate data</u>. Pacific Grove, CA, Thomson Brooks/Cole.

Li, L. P., Weinberg, C. R., Darden, T. A. and Pedersen, L. G. (2001). "Gene selection for sample classification based on gene expression data: study of sensitivity to choice of parameters of the GA/KNN method." <u>Bioinformatics</u> **17**(12): 1131-1142.

Liu, J. J., Cutler, G., Li, W., Pan, Z., Peng, S., Hoey, T., Chen, L. and Ling, X. B. (2005). "Multiclass cancer classification and biomarker discovery using GA-based algorithms." <u>Bioinformatics</u> **21**(11): 2691-7.

Mcculloch, W. S. and Pitts, W. (1990). "A Logical Calculus of the Ideas Immanent in Nervous Activity (Reprinted from Bulletin of Mathematical Biophysics, Vol 5, Pg 115-133, 1943)." <u>Bulletin of Mathematical Biology</u> **52**(1-2): 99-115.

Moore, A. (2001). Statistical Data Mining Tutorials. Support Vector Machines., <u>http://www-2.cs.cmu.edu/~awm/tutorials/</u>. **2001**.

Ooi, C. H. and Tan, P. (2003). "Genetic algorithms applied to multi-class prediction for the analysis of gene expression data." <u>Bioinformatics</u> **19**(1): 37-44.

Peng, S., Xu, Q., Ling, X., Peng, X., Du, W. and Chen, L. (2003). "Molecular classification of cancer types from microarray data using the combination of genetic algorithms and support vector machines." <u>FEBS Lett.</u> **555**(2): 358-62.

Sha, N. J., Vannucci, M., Tadesse, M. G., Brown, P. J., Dragoni, I., Davies, N., Roberts, T. R. C., Contestabile, A., Salmon, M., Buckley, C. and Falciani, F. (2004). "Bayesian variable selection in multinomial probit models to identify molecular signatures of disease stage." <u>Biometrics</u> **60**(3): 812-819.

Smola, A. J. (2000). <u>Advances in large margin classifiers</u>. Cambridge, Mass., MIT Press.

Tabachnick, B. G. and Fidell, L. S. (2001). <u>Using multivariate statistics</u>. Boston, MA, Allyn and Bacon.

Tibshirani, R., Hastie, T., Narasimhan, B. and Chu, G. (2002). "Diagnosis of multiple cancer types by shrunken centroids of gene expression." <u>Proc Natl Acad Sci U S A. Jan 20;</u> **99**(10): 6567-72.

Ye, N. (2003). <u>The handbook of data mining</u>. Mahwah, N.J.; London, Lawrence Erlbaum Assoc.

Yeoh, E. J., Ross, M. E., Shurtleff, S. A., Williams, W. K., Patel, D., Mahfouz, R., Behm, F. G., Raimondi, S. C., Relling, M. V., Patel, A., Cheng, C., Campana, D., Wilkins, D., Zhou, X. D., Li, J. Y., Liu, H. Q., Pui, C. H., Evans, W. E., Naeve, C., Wong, L. S. and Downing, J. R. (2002). "Classification, subtype discovery, and prediction of outcome in pediatric acute lymphoblastic leukemia by gene expression profiling." <u>Cancer Cell</u> **1**(2): 133-143.

Vaquerizas J.M., Conde L., Yankilevich P., Cabezon A., Minguez P., Diaz-Uriarte R., Al-Shahrour F., Herrero J & Dopazo J. (2005), Gepas an experiment-oriented pipeline for the analysis of microarray gene expression data. *Nucleic Acids Research* 33 (Web Server issue):W616-W620.