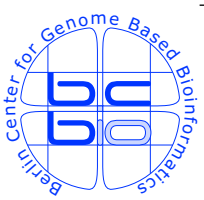

Max Planck Institute for Molecular Genetics
Computational Diagnostics Group @ Dept. Vingron
Ihnestrasse 63-73, D-14195 Berlin, Germany
http://compdiag.molgen.mpg.de/

Structured Analysis of Microarrays

User's Guide to the Bioconductor package *stam*

Claudio Lottaz[1] **and Rainer Spang**

Max Planck Institute for Molecular Genetics
Computational Diagnostics, Department for Computation Molecular Biology
Ihnestr. 63-73, D-14195 Berlin, Germany

Technical Report
Nr. 2004/03

Abstract

This is the vignette of the Bioconductor compliant packege `stam`. We describe our implementation of *structured analysis of microarray data* for decomposing complex clinical phenotypes in clinical gene expression profiling studies.

[1]Corresponding author: `claudio.lottaz@molgen.mpg.de`

Contents

Chapter 1

Introduction

In clinical context microarray data can be used in a straight forward manner for diagnostic tasks. Thereby, standard classification methods developed in the machine learning community have been suggested and applied in clinical research to distinguish between clinically relevant phenotypes. However, it is commonly accepted that such phenotypes are often complex, i.e. caused by different causes. Thus we expect to find heterogenous gene expression profiles in patients with homogenous clinical phenotype.

In order to decompose complex phenotypes, we suggest a procedure we call *structured analysis of Microarrays*. In this procedure we train many classifiers to recognize the same complex phenotype. Each classifier focusses on a single biological aspect by extracting only the expression data of the corresponding genes. Many of these classifiers will be unable to recognize the phenotype of interest and are thus discarded. Others will recognize a subset of patients and some of them hopefully discover all patients. *Structured analysis of microarrays* uses classifiers with high specificity, we call them molecular symptoms, to further stratify patients on a molecular level. Patterns of absence and presence of molecular symptoms identify particular groups of patients. Due to the biological focus of our classifiers, molecular symptoms always have a biological meaning.

The Bioconductor compliant R package *stam* described in this document implements *structured analysis of microarrays* based on the Gene Ontology [1]. GO terms attributed to leaf nodes of the GO graph are used as biological focusses for potential molecular symptoms. The hierarchy of the Gene Ontology is used to compute more general classifiers by weighted averaging. Thereby, good classifiers obtain higher weights. A shrinkage process of these weights, similar to the one used in PAM [4] to select genes, restricts the graph of classifiers in a cross validation setting to the branches linked to the phenotype.

The remainder of this document describes the usage of the *stam* package. *stam* relies on Bioconductor meta data packages to implement structured training of classifiers (Section 2) and structured prediction (Section 3). For further convenience we have implemented a complete structured evaluation of a data set (Section 4) and a web based possibility to

explore some crucial parameters of such an analyses interactively (Section 5).

Chapter 2

Training Classifiers

The training of a structured classifier for a phenotype in microarray data consists of the following steps:

- Generate a classifier network according to the chip's annotations and the GO hierarchy.
- Perform cross-validated PAM fits in each leaf node to determine adequate PAM shrinkage levels and compute performance for several graph shrinkage candidates.
- Compute single structured model using the best graph shrinkage level chosen by the user or automatically according some performance criterion calculated in the cross validation step.

Before starting any analysis you have to load the *stam* package in your R session as follows:

```
> library(stam)
```

```
Loading required package: GO
Loading required package: pamr
Loading required package: cluster
Loading required package: annaffy
Loading required package: KEGG
```

For illustration of *stam* usage we use the Golub data set on acute leukemia [3] as it is stored in the *golubEsets* data package. For normalization we use Bioconductor's *vsn* package implementing the variance stabilization and calibration method [2]. For preparing the data, issue the following commands:

```
> library(vsn)
> library(golubEsets)
```

```
> data(golubMerge)
> golubNorm <- vsn(golubMerge, describe.preprocessing = FALSE,
+     verbose = FALSE)
> golubTrain <- golubNorm[, 1:38]
> golubTest <- golubNorm[, 39:72]
```

2.1 Generate a raw classifier graph

The raw classifier graph can be generated by the function `stam.net`. This function needs the name of the chip used in the study to be analyzed as well as the root node to be used for the classifier graph. The function returns an object of class *stamNet*.

```
> net <- stam.net(chip = "hu6800", root = "GO:0005576",
+     probes = rownames(exprs(golubMerge)))
```

The string given as the chip's name is used to load the annotation data. Thus *stam* expects a library of the same name to be installed, where it looks for the `<chip-name>GO` hash as it is provided by Bioconductor meta data packages. The root of the classifier graph must be identified by a string interpreted as a GO identifier of the form '`GO:ddddddd`'. The default is `GO:0008150` which represents the 'biological process' branch of the Gene Ontology. Other prominent candidates may be `GO:0003674` (molecular function) or `GO:0005575` (cellular component). However, any valid GO identifier is allowed.

The classifier graph may be analysed using the print function as is shown in the next chunk of R code. Printing the object returned by `stam.net` directly shows some properties of the generated classifier graph. One component of this object holds an entry for each node. Summary information on each node can also be printed as shown below.

```
> print(net)
```

```
stamNet on chip hu6800
Chip data package:
  version 1.8.10
   Fri Oct 7 14:59:08 2005; tliu
Gene Ontology GO package:
  version 1.8.10
   Fri Oct 7 14:57:00 2005; tliu)
root is set to
GO:0005576 (extracellular region)
holds 46 nodes, 33 of which are leafs and 13 are inner nodes
870 of 7129 probesets have annotations (12.2%)
```

```
> print(net@nodes[[31]])
```

```
stamLeaf GO:0005605 basal lamina
from category cell components
contains 4 genes
[1] "L31801_at" "M20471_at" "M22005_at" "X99584_at"
```

```
> print(net@nodes[["GO:0005579"]])
```

```
stamLeaf GO:0005579 membrane attack complex
from category cell components
contains 8 genes
[1] "X79234_at"        "U44755_at"        "HG2614-HT2710_at"
[4] "X72925_at"        "L10035_at"        "M30773_at"
[7] "U85265_at"        "X63337_at"
```

A convenient meands to explore the information on the raw classifier graph is provided by the function `stam.writeHTML` which, applied on a *stamNet* object, writes a set of interlinked HTML pages.

```
> stam.writeHTML(net)
```

One referrable section of an HTML page is written for each node. Each such section contains links to the parents and the children. Sections for leaf nodes contain links to the Affymetrix annotations of the genes they contain.

2.2 Cross validate graph shrinkage candidates

In order to choose the graph shrinkage level reasonnably, we provide a cross validation method similar to the procedure suggested for choosing shrinkage fore gene selection in PAM. `stam.cv` applies this method and returns an object of class *stamCV*:

```
> golubTrain.cv <- stam.cv(golubTrain, "ALL.AML", chip = "hu6800",
+     root = "GO:0005575", ndeltas = 10)
```

This call also generates the above mentioned raw classifier graph. The considered shrinkage candidates can be provided to this function directly. If only a number of candidates is specified using `ndeltas` `stam.cv` chooses these equidistantly within the range of performance measures observed in the leaf nodes.

For further investigation of the returned *stamCV* object you can use the *print* and *plot* methods.

```
> print(golubTrain.cv)
```

```
stamCV on expression matrix holding 38 samples from 2 classes.
Structure from: stamNet on chip hu6800
Chip data package:
  version 1.8.10
   Fri Oct 7 14:59:08 2005; tliu
Gene Ontology GO package:
  version 1.8.10
   Fri Oct 7 14:57:00 2005; tliu)
root is set to
GO:0005575 (cellular_component)
holds 522 nodes, 356 of which are leafs and 166 are inner nodes
5520 of 7129 probesets have annotations (77.4%)
Class priors: 'ALL': 0.631579 'AML': 0.3684211
Class weights: 'ALL': 0.631579 'AML': 0.3684211
Shrinkage candidates: 0 0.1334701 0.2669402 0.4004102 0.5338803 0.6673504 0.8008205 0.934

Results from 10-fold cross validation:
      Root.error.rate Root.performance Mean.redundancy Nodes Genes
0             0.02632          0.58335         0.85135   290  5418
0.133         0.02632          0.52824         0.88645   244  5405
0.267         0.02632          0.46613         0.98871   157  5316
0.4           0.02632          0.42174         1.12954   106  5158
0.534         0.02632          0.36226         1.34944    75  4802
0.667         0.02632          0.28298         1.88620    49  3845
0.801         0.02632          0.18054         3.17959    33  2271
0.934         0.02632          0.15729         4.25262    27  2016
1.068         0.02632          0.14331         5.65678    23  1822
```

```
> plot(golubTrain.cv, delta = 0.6)
```

The *plot* method can actually provide two types of plots as shown below. One of them shows the error rate and performance measure in the root node as well as the mean redundancy across the resulting classifier graph depending on the graph shrinkage level (Figure 2.1). Thereby, our performance measure in the root node is similar to a deviance and thus to be minimized. The second plot shows number of nodes and accessible genes in the remaining graphs depending on the graph shrinkage level (Figure 2.2).

In order to investigate the cruss validation results, you can also apply the function `stam.writeHTML` on the object returned by `stam.cv`.

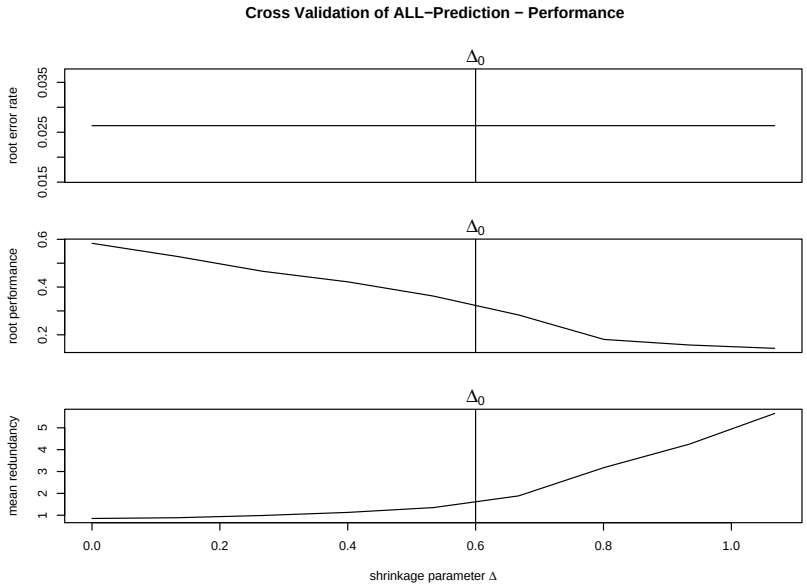

Figure 2.1: Results of cross validation - performance and redundancy

2.3 Computing a single model

After deciding for the best graph shrinkage level based on the cross validation results, we suggest to determine a single model to recognize the investigated phenotype. This task is performed using the `stam.fit` function. The graph shrinkage level can be chosen through the `delta` argument. It can also be determined automatically based on a weighting factor between performance and redundancy specified in `alpha`. If `alpha` is set to `NULL` the error rate in the root node is minimized. You can also provide a sequence of weighting factors between 0 and 1 in `alpha`. In this case `stam.fit` computes evaluation criteria for each of these factors and asks the user to provide the best graph shrinkage level interactively.

```
> golubTrain.fit <- stam.fit(golubTrain.cv, golubTrain,
+     alpha = seq(0, 1, 0.1))

 - Fitting for best shrinkage...
    Best Delta Root Error Rate Performance Redundancy #Nodes #Genes
0       0.0000          0.0263      0.5833     0.8514    290   5418
0.1     0.1335          0.0263      0.5282     0.8865    244   5405
0.2     0.2669          0.0263      0.4661     0.9887    157   5316
0.3     0.5339          0.0263      0.3623     1.3494     75   4802
```

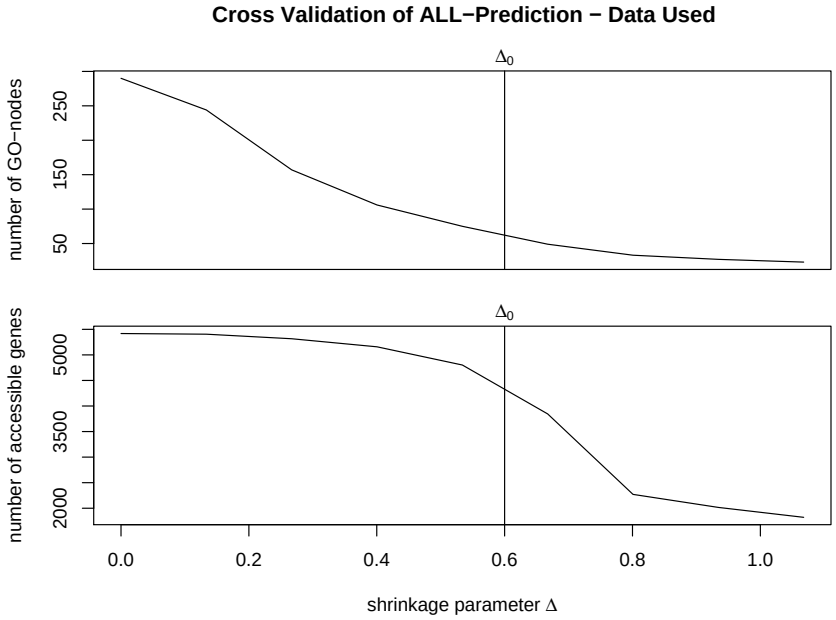

Figure 2.2: Results of cross validation - number of nodes and genes

```
0.4     0.6674          0.0263      0.2830      1.8862      49   3845
0.5     0.6674          0.0263      0.2830      1.8862      49   3845
0.6     0.8008          0.0263      0.1805      3.1796      33   2271
0.7     0.8008          0.0263      0.1805      3.1796      33   2271
0.8     0.8008          0.0263      0.1805      3.1796      33   2271
0.9     0.9343          0.0263      0.1573      4.2526      27   2016
1       1.0678          0.0263      0.1433      5.6568      23   1822
    . fitting weights with shrinkage 0.6674 ...
      yields error rate: 0.02631579 (performance 0.2829428, redundancy 1.886352)
      through 49 nodes with 3845 accessible genes
```

For further investigation we provide *print* and *plot* methods on the *stamFit* object returned by `stam.fit`. The *plot* method on *stamFit* objects generates two different pluts. The first plut, only generated when a set of weighting factors is given to `stam.fit`, illustrates the dependency between weighting factor and resulting graph shrinkage (Figure 2.3). The second plot shows the nodewise evaluation of classifiers according to sensitivity, specificity, redundancy and performance (Figure 2.4).

```
> print(golubTrain.fit)
```

```
stamFit to predict from 2 classes.
Structure from: stamNet on chip hu6800
Chip data package:
  version 1.8.10
   Fri Oct 7 14:59:08 2005; tliu
Gene Ontology GO package:
  version 1.8.10
   Fri Oct 7 14:57:00 2005; tliu)
root is set to
GO:0005575 (cellular_component)
holds 49 nodes, 17 of which are leafs and 32 are inner nodes
5520 of 7129 probesets have annotations (77.4%)
Class priors: 'ALL': 0.6316 'AML': 0.3684
Class weights: 'ALL': 0.6316 'AML': 0.3684
Shrinkage parameter: 0.6674  chosen by considering weighted scores between root performan
Candidates for performance weights: 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1
Left  49  nodes with weights
           Performance Redundancy ALL (sens.) AML (sens.) ALL (spec.)
GO:0005578      0.5325     1.5071      0.9167      1.0000      1.0000
GO:0005578      0.5325     1.5071      0.9167      1.0000      1.0000
GO:0005576      0.5371     1.4969      0.9167      1.0000      1.0000
GO:0008305      0.6360     1.4493      0.9583      0.8571      0.8571
GO:0005887      0.2082     3.4595      0.9583      0.9286      0.9286
GO:0005887      0.2024     3.2692      0.9583      0.9286      0.9286
GO:0031226      0.2024     3.2692      0.9583      0.9286      0.9286
GO:0005886      0.5136     1.3966      0.9583      1.0000      1.0000
GO:0005886      0.2540     1.9886      0.9583      0.9286      0.9286
GO:0016021      0.2024     3.2692      0.9583      0.9286      0.9286
GO:0031224      0.2024     3.2692      0.9583      0.9286      0.9286
GO:0016020      0.2247     2.1914      0.9583      0.9286      0.9286
GO:0005795      0.3219     1.8469      0.9167      1.0000      1.0000
GO:0005794      0.3219     1.8469      0.9167      1.0000      1.0000
GO:0005764      0.1202     2.4559      1.0000      1.0000      1.0000
GO:0005764      0.1202     2.4559      1.0000      1.0000      1.0000
GO:0005773      0.1202     2.4559      1.0000      1.0000      1.0000
GO:0008091      0.5395     1.3947      1.0000      0.8571      0.8571
GO:0030864      0.5395     1.3947      1.0000      0.8571      0.8571
GO:0030863      0.5395     1.3947      1.0000      0.8571      0.8571
GO:0005938      0.5395     1.3947      1.0000      0.8571      0.8571
```

GO:0005829	0.3158	1.7920	1.0000	0.9286	0.9286
GO:0005829	0.3442	1.7666	0.9583	0.9286	0.9286
GO:0005737	0.2160	2.2747	0.9583	0.9286	0.9286
GO:0005737	0.2393	1.8303	1.0000	0.9286	0.9286
GO:0015629	0.4910	1.4381	1.0000	1.0000	1.0000
GO:0015629	0.5111	1.3704	1.0000	1.0000	1.0000
GO:0005882	0.5269	1.4326	1.0000	0.8571	0.8571
GO:0005882	0.5269	1.4326	1.0000	0.8571	0.8571
GO:0005856	0.5379	1.3082	1.0000	1.0000	1.0000
GO:0005856	0.5272	1.3325	1.0000	1.0000	1.0000
GO:0005634	0.5543	1.2942	0.9583	0.9286	0.9286
GO:0005634	0.6146	1.2819	0.9167	0.8571	0.8571
GO:0043231	0.2380	1.8061	1.0000	1.0000	1.0000
GO:0043232	0.5272	1.3325	1.0000	1.0000	1.0000
GO:0043229	0.3016	1.6377	1.0000	1.0000	1.0000
GO:0005622	0.4724	1.6947	1.0000	0.7857	0.7857
GO:0005622	0.3494	1.5903	1.0000	0.9286	0.9286
GO:0005624	0.1150	3.0216	0.9583	0.9286	0.9286
GO:0005624	0.1150	3.0216	0.9583	0.9286	0.9286
GO:0005625	0.6018	1.3879	0.9167	0.7857	0.7857
GO:0000267	0.1499	2.3509	0.9583	0.9286	0.9286
GO:0005623	0.2050	1.9628	1.0000	0.9286	0.9286
GO:0043227	0.2380	1.8061	1.0000	1.0000	1.0000
GO:0043226	0.3016	1.6377	1.0000	1.0000	1.0000
GO:0008372	0.1971	2.2253	0.9583	0.9286	0.9286
GO:0043235	1.2356	0.9541	0.9583	0.2143	0.2143
GO:0043234	1.2282	0.9920	0.9167	0.2143	0.2143
GO:0005575	0.2829	1.7422	1.0000	0.9286	0.9286

	AML (spec.)
GO:0005578	0.9167
GO:0005578	0.9167
GO:0005576	0.9167
GO:0008305	0.9583
GO:0005887	0.9583
GO:0005887	0.9583
GO:0031226	0.9583
GO:0005886	0.9583
GO:0005886	0.9583
GO:0016021	0.9583

```
GO:0031224      0.9583
GO:0016020      0.9583
GO:0005795      0.9167
GO:0005794      0.9167
GO:0005764      1.0000
GO:0005764      1.0000
GO:0005773      1.0000
GO:0008091      1.0000
GO:0030864      1.0000
GO:0030863      1.0000
GO:0005938      1.0000
GO:0005829      1.0000
GO:0005829      0.9583
GO:0005737      0.9583
GO:0005737      1.0000
GO:0015629      1.0000
GO:0015629      1.0000
GO:0005882      1.0000
GO:0005882      1.0000
GO:0005856      1.0000
GO:0005856      1.0000
GO:0005634      0.9583
GO:0005634      0.9167
GO:0043231      1.0000
GO:0043232      1.0000
GO:0043229      1.0000
GO:0005622      1.0000
GO:0005622      1.0000
GO:0005624      0.9583
GO:0005624      0.9583
GO:0005625      0.9167
GO:0000267      0.9583
GO:0005623      1.0000
GO:0043227      1.0000
GO:0043226      1.0000
GO:0008372      0.9583
GO:0043235      0.9583
GO:0043234      0.9167
GO:0005575      1.0000
```

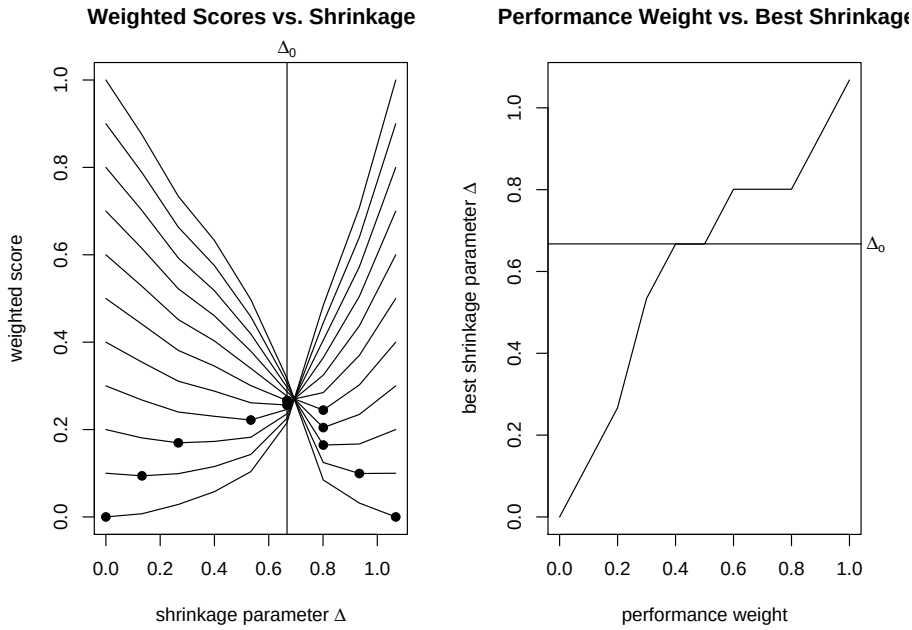

Figure 2.3: Results of model fit - weighting vs. shrinkage

```
> plot(golubTrain.fit)
```

The function `stam.writeHTML` can also be applied on *stamFit* objects. If the package `graphviz` is installed on your system, an HTML page with a clickable graph plot is generated for exploration of the fitted model. The function `stam.graph.plot` generates this graph layout using the `dot` utility from `graphviz` (Figure 2.5.

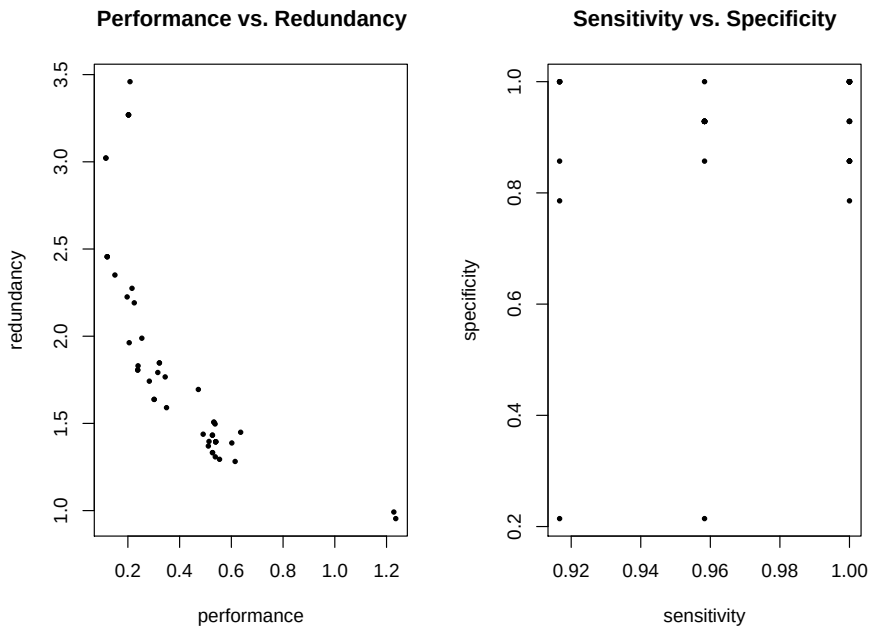

Figure 2.4: Results of model fit - nodewise evaluation

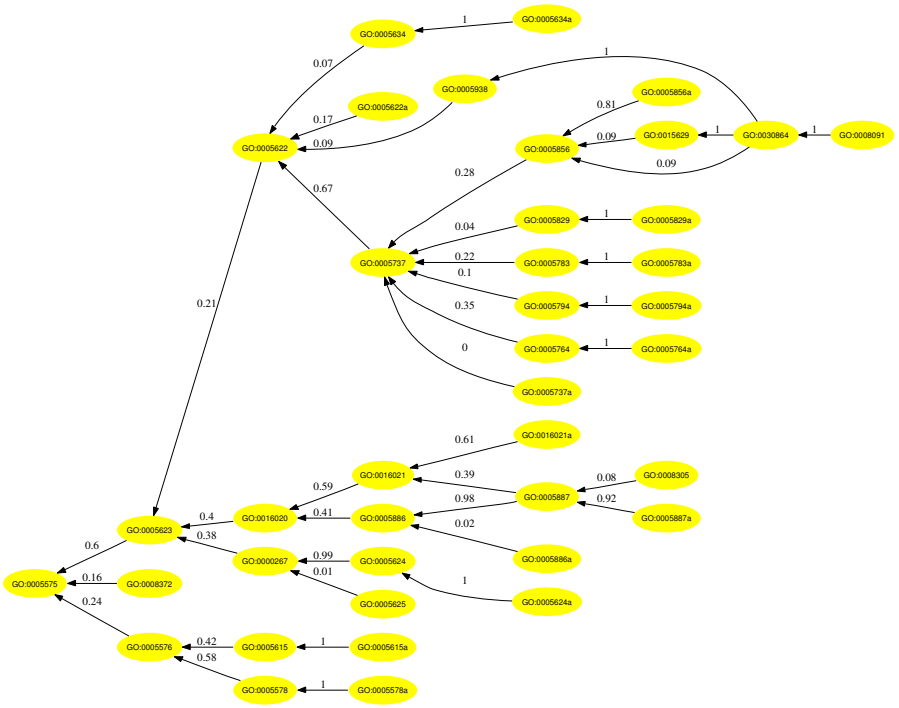

Figure 2.5: Shrunken classifier graph

Chapter 3

Structured Prediction

The model generated in the above described manner is meant to be used as a classifier for new expression profiles. The `stam.predict` function allows for this task and returns an object of class *stamPrediction*. When calling `stam.predict` the user can specify a series of expression profiles and may specify a number of these to be part of the test set while the others are assumed to be from the training set. The two variants of the call to the structured prediction function are shown here:

```
> golubTest.pred <- stam.predict(golubTrain.fit, golubTest,
+     pData(golubTest)[, "ALL.AML"])

 - Predicting new samples...

> golubMerge.pred <- stam.predict(golubTrain.fit, golubNorm,
+     pData(golubNorm)[, "ALL.AML"], testset = 39:72)

 - Predicting new samples...
```

The `stam.predict` returns an object of class *stamPrediction*. In addition to the usual call to `stam.writeHTML` the *print*, *plot*, and *image* methods may be used for further exploration of the structured prediction:

```
> print(golubTest.pred)

stamPrediction on 34 samples.
Root predictor says:
   X1    X2    X3    X4    X5    X6    X7    X8    X9   X10   X11
"ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
  X12   X13   X14   X15   X16   X17   X18   X19   X20   X21   X22
"ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL" "ALL"
  X23   X24   X25   X26   X27   X28   X29   X30   X31   X32   X33
```

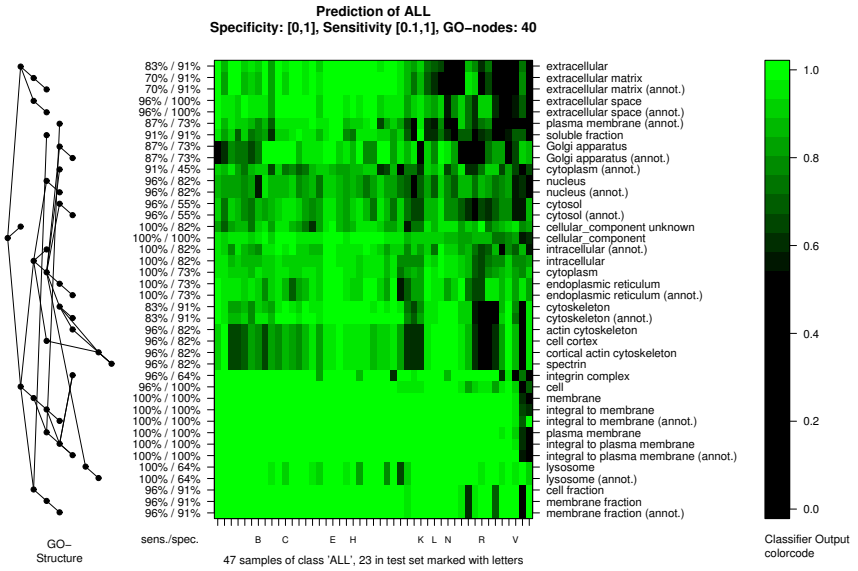

Figure 3.1: Molecular symptoms image of a prediction, test samples are marked with capitals.

```
"ALL" "AML" "AML" "AML" "AML" "AML" "ALL" "AML" "AML" "AML" "ALL"
  X34
"AML"
```

```
> plot(golubTest.pred)
> image(golubMerge.pred)
```

Structured prediction means to compute prediction probabilities for each considered phenotype as well as each expression profile of interest. Thereby, the classifier in each node of the shrunken classifier graph is evaluated and the averaging through the node weights computed in the model fit step is performed. Thus we can derive an overall decision for new expression profiles according to the root node classeifier. Additionally, we can consider the pattern of absence and presence of molecular symptoms for each node to define a similarity to expression profiles used in training. This is illustrated using the *image* method in Figure 3.1.

The `stam.writeHTML` function can generate a clickable map for the molecular symptom image, where each of the GO terms can be clicked in order to view the results of the corresponding classifier. The *plot* method illustrates nodewise sensitivity, specificity, redundancy and performance.

Chapter 4

All in One

The function `stam.evaluate` calls the functions mentioned so far in sequence by the function in order to performe a complete structured analysis. Thereby, we have two application paradigms in mind: the predictive usage and the explorative usage.

4.1 Explorative Use

When using structured analyses of microarray in an explorative manner, we do not split the available expression profiles into training and test set. Thus we are not able to evaluate the prediction accuracy of the classifier. However, we are looking for particular structure in the classifiers of the shrunken classifier graph. In order to call `stam.evaluate` exploratively it must be called with the argument `testset` set to `NULL`:

```
> golubNorm.eval.explore <-
+                   stam.evaluate(golubNorm, "ALL.AML", testset = NULL,
+                                 chip="hu6800", root="GO:0005576",
+                                 alpha=seq(0, 1, 0.1), ndelta=10)
```

For further investigation of the returned *stamEval* object, a print method is provided but we recommend the usage of `stam.writeHTML`. The latter function generates a series of inter-linked HTML pages holding most plots and both clickable maps of the generated objects. Furthermore, the *stamEval* object holds the original objects returned by `stam.cv`, `stam.fit`, and `stam.predict` in its slots. The user can further investigate using the corresponting *print*, *plot*, and *image* methods.

4.2 Predictive Use

The predictive use of structured analysis of microarrays is the standard behavior of `stam.evaluate`. By default a third of the expression profiles is chosen randomly as testset

while the rest is used for training. The user can explicitly set the testset through the `testset` parameter of `stam.evaluate`:

```
> golubNorm.eval.predict <-
+                   stam.evaluate(golubNorm, "ALL.AML", testset=39:72,
+                                 chip = "hu6800", root = "GO:0005576",
+                                 ndelta = 10)
```

The same remarks as above apply according the further investigation possibilities.

Chapter 5

StAM WWW-Server

For the interactive exploration of the graph shrinkage level and a few parameters to influence the model graph plot as well as the molecular symptoms image, we have implemented a WWW based solution. The function `stam.writeHTML` can write HTML forms for these parameters directly into the HTML pages representing the analysis results. We provide CGI scripts with the *stam* package which collect the user entries from these forms and store them into requests stored in a temporary directory writable for the web server. The `stam.serve` provides the functionality to check this temporary directory and perform the stored requests as illustrated in Figure 5.1. The WWW client is redirected to a progress page which reloads automatically every other second. As soon as `stam.serve` has finished treating the request the progress page redirects the browser to the new result page.

When the `stam.serve` function is first called after installing the *package*, the location of the temporary directory for requests, the directory for CGI scripts and the URL where these scripts can be found is obtained either from function arguments or interactively.

```
> stam.serve(tmp.path = "/home/upload/stam",
+            cgi.path = "/home/cgi-bin/stam",
```

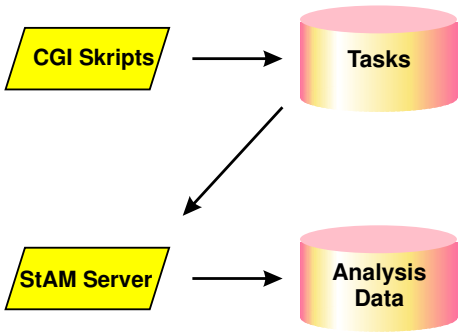

Figure 5.1: Architecture of StAM's server feature.

```
+            cgi.url  = "/cgi-bin/stam")

The CGI URL has changed, make sure to regenerate the HTML pages
to be worked with through the server.
 - start listening '/home/upload/stam'...
   Stop the server by creating '/home/upload/quit'.
   Make sure you have written the HTML pages to be worked with
   through the StAM server with 'options(stam.write.forms=TRUE)'
```

`stam.serve` stores this information into the package installation and copies the modified CGI scripts to the given location. Now `stam.serve` starts "listening" in the temporary directory and will do so without the installation steps in subsequent calls.

In order to use *stam*'s server feature, you need to set the option `stam.write.forms` to `TRUE` before rewriting the HTML code for the model you want to work with:

```
> options(stam.write.forms=TRUE)
> stam.writeHTML(golubNorm.eval.explore)
```

Now, if you load the generated HTML page into your browser, you will see the forms to enter modified parameters. It is enough to start `stam.serve` in order to use these forms interactively.

Acknowledgements

This research has been supported by BMBF grant 031U117/031U217 of the German Federal Ministry of Education and the National Genome Research Network.

Chapter 6

Bibliography

[1] The Gene Ontology Consortium. Gene ontology: Tool for the unification of biology. *Nature Genetics*, 25:25–29, May 2000.

[2] W. Huber, A. von Heydebreck, H. Sültmann, A. Poustka, and M. Vingron. Variance stabilization applied to microarray data calibration and to the quantification of differential expression. *Bioinformatics* vol. 18, suppl. 1, pp. S96-S104, 2002.

[3] T.R. Golub, D.K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J.P. Mesirov, H. Coller, M.L. Loh, J.R. Downing, M.A. Caligiuri, C.D. Bloomfield, and E.S. Lander. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science*, vol. 286, pp. 531-537, 1999.

[4] R. Tibshirani, T. Hastie, B. Narashiman, and B. Chu. Diagnosis of multiple cancer types by shrunken centroids of gene expression. *PNAS*, 99:6567–6572, May 2002.