Gene Expression Variation Analysis (GEVA)

Itamar José Guimarães Nunes[1], Murilo Zanini David[2],
Bruno César Feltes[1], and Marcio Dorn[1]

April 25, 2023

Contents

[1]Structural Bioinformatics and Computational Lab (SBCB) – Federal University of Rio Grande do Sul (UFRGS)

[2]Graduate Program in Ecology – Federal University of Rio Grande do Sul (UFRGS)

1 Introduction

`GEVA` is a package for the analysis of differential gene expression in multiple experimental comparisons. It takes into account the fold-changes and p-values from previous differential expression (DE) results that use large-scale data (*e.g.*, microarray and RNA-seq) and evaluates which genes would react in response to the distinct experiments. This evaluation involves an unique pipeline of statistical methods, including weighted summarization, quantile detection, cluster analysis, and ANOVA tests, in order to classify a subset of relevant genes whose DE is similar or dependent to certain biological factors.

This guide introduces the basic usage of `geva` package and focuses on its main features to perform the entire analysis from the input to the final classification. However, for more detailed specifications regarding classes, functions, and arguments from `geva`, please check the "Reference Guide" available in our GitHub repository. Alternatively, the local documentation can be accessed by typing `?geva` in the R console.

Before proceeding to the current methodology, it is assumed that the user already knows how to manipulate datasets and perform DE analyses using Bioconductor packages or any external tool that is capable to produce results from DE comparisons. For users with less familiarity about this

subject, please read the tutorials described to the available R packages for DE analyses, such as *limma* [1] for microarrays and *DESeq2* [2] for RNA-seq. In addition, some standalone applications employ the equivalent methods from R to achieve the same results, including GEAP (for microarrays) [3] and Chipster (for RNA-seq) [4], both of which provide a graphical user interface and do not require programming knowledge.

2 Installation

This package is available on GitHub and can be installed through the following command:

```
BiocManager::install("geva")
```

Note that this command requires the *BiocManager* package (installed via `install.packages('BiocManager')`). After downloading and installing the sources, use the following command to load `geva` from the local package library:

```
library(geva)
```

3 Data input

The input data is essentialy two or more tables produced by DE analyses that include logFC and (adjusted) p-value columns in association to the genes (row names). For microarrays, particularly, the probes may be used as row names along with a Gene Symbol column, which can be attached to the final results at the end of the analysis. Moreover, although only two tables are required for GEVA's minimal usage, the inclusion of several columns is strongly recommended to achieve a resonable statistical precision. Note that experiments can be grouped and analyzed in multiple contexts at once in

GEVA, and likewise in this case, each group should include several experiments to attain better results from the statistical tests.

GEVA gives some data input alternatives so that users can provide objects from the local R environment or from external table files. These alternatives are described in the sub-sections below, whereas only one of them is required to accomplish the same desired output.

3.1 Alternative 1 – Tab-delimited Text Files

Programs that feature DE analysis usually output a table of DE results which is exported as a plain text file. By convention, the saved file should be formatted as one row per line and one tab-delimited value per column, but other formats may be used as well. For the conventional format, the `geva.read.tables` function can be called using default parameters as demonstrated below:

```
# Replace the file names below with actual file paths
filenms <- c("cond_A_2h.txt", "cond_B_2h.txt", "cond_C_2h.txt",
             "cond_A_4h.txt", "cond_B_4h.txt", "cond_C_4h.txt")
ginput <- geva.read.tables(filenms)
```

The code above will produce a `GEVAInput` object, which stores all the relevant information regarding the input. It reads each file as a table by calling `read.table` internally and extracting the columns containing `logFC` and `adj.P.Val` columns, then merging all columns into two tables (one for *logFC* values and one for weights).

In addition, the `geva.read.tables` function has some handful optional parameters to be considered. For instance, if the `dirname` parameter is used instead of `filenames`, all files inside the directory `dirname` that match the pattern given by the `files.pattern` argument (default is `"\\.txt$"` or TXT files) will be included. Other relevant arguments are `col.values` (by default, `"logFC"`) and `col.pvals` (by default, `"adj.P.Val"`), used to

indicate which columns names are used for *logFC* and (adjusted) p-values. Vectors of multiple `character` elements can be passed to these arguments if the column names differ among the table files so that the first matching column is included. Furhermore, if one wants to append additional columns in the analysis (*e.g.*, gene names or gene symbols) to associate them to the final results, the column names can be specified at the `col.other` argument.

3.2 Alternative 2 – Multiple table objects

Table objects, particularly of `matrix` and `data.frame` types, can be used as input to GEVA as long as they include the *logFC* and p-value columns. The `geva.merge.input` function receives two or more table arguments and extracts their corresponding columns to include in the final merge. For example, given two `data.frame` objects defined as `dt1` and `dt1` in the global environment, the command for this step becomes:

```
# dt1 and dt2 are examples of input data.frames
# containing logFC and adj.P.Val columns
ginput <- geva.merge.input(dt1, dt2)
```

The code above will produce a `GEVAInput` object, which stores all the relevant information regarding the input. Arguments are passed individually and can also be named to define the columns in the final merge (*e.g.*, `cond1=dt1, cond2=dt2` to append the extracted columns as `"cond1"` and `"cond2"`). Note that some arguments from `geva.read.tables`, including `col.values`, `col.pvals`, and `col.other`, have the same principle as in `geva.merge.input`[1] (see *Alternative 1*).

[1]Actually, `geva.read.tables` calls internally the `geva.merge.input` function upon reading each table file.

3.3 Alternative 3 – Results from *limma*

If the DE analysis is being performed from a specific R package such as *limma*, the results can be converted to a `matrix` or `data.frame` and passed as arguments to `geva.merge.input` as demonstrated in the previous section (see *Alternative 2*). For example, if *limma* was used to produce two `MArrayLM` objects (*i.e.*, DE results using linear model fit), these can be converted to `data.frame` using `limma::topTable`, then passed to `geva.merge.input` as demonstrated below:

```
# malm1 and malm2 are MArrayLM objects produced by
# limma (e.g., using eBayes)
dt1 <- topTable(malm1, number=999999, sort.by="none")
dt2 <- topTable(malm2, number=999999, sort.by="none")
ginput <- geva.merge.input(dt1, dt2)
```

The code above will produce a `GEVAInput` object, which stores all the relevant information regarding the input. Since both `dt1` and `dt2` already include `"logFC"` and `"adj.P.Value"` columns, `geva.merge.input` can be called using the defaults parameters.

3.4 Alternative 4 – Ideal input data (for tests only)

Be it due the abscence of experimental data or merely for didatical reasons, there may be some situations where the features in this package have to be immediately accessed and tested without needing to provide any real data, since two or more DE analyses must be performed before using GEVA. In this sense, the `geva.ideal.example` function can be used to generate a random input that simulates real processed inputs by GEVA. The function is called as follows:

```
# (optional) Sets the initial seed to reproduce the
# results presented here
set.seed(1)
# Generates a random GEVAInput with 10000 probes
# and 6 columns
ginput <- geva.ideal.example()
```

The code above will generate a `GEVAInput` object with random values within a normal distribution and some random outliers to simulate the relevant results. In addition, all columns are grouped into experimental condition groups (*factors*) so that *factor-dependent* and *factor-specific* results could be produced by the end of the analysis test. Note that although the output is essentially "random", the same result can be reproduced by using `set.seed` before `geva.ideal.example`.

4 Input data post-processing (optional)

Considering that the final results will strongly depend on the input values in the concatenated tables, some tweaks in the obtained `GEVAInput` can be done to improve them in terms of statistics and presentation. Some features implemented in GEVA that allow this kind of post-processing of the `GEVAInput` objects are presented over the following sub-sections.

4.1 Numeric table correcting

First off, one may want to eliminate primary sources of errors from the numeric tables before proceeding to the next steps. The calculations become prone to bias when missing values (`NA`) or infinite numbers (`Inf`) are present, so except in rare cases where their inclusion is intentional, removing them is a reasonable choice.

In this sense, the `geva.input.correct` function will remove all missing (`NA`), not-a-number (`NaN`), and infinite (`Inf` or `-Inf`) values from `GEVAInput` upon calling the following command:

```
# Removes the rows containing missing and infinite values
ginput <- geva.input.correct(ginput)
```

The validation is only applied to the numeric tables in `GEVAInput` (*i.e.*, `@values` and `@weights` slots). As a result, if any invalid values were found, their rows are removed. However, there is an exceptional case where one column is entirely made of invalid values which would cause all rows to be marked as invalid, so `geva.input.correct` removes such columns in advance to prevent the exclusion of the entire table.

4.2 Filtering values below statistical significance

The `GEVAInput` stores a table of transformed p-values as weights (`@weights` slot, called by `inputweights` function) employed in some calculations during the summarization step (discussed in the next section). While the inclusion of weights is used to minimize statistical errors, it also follows the assumption that all rows have at least one significant p-value. In this sense, the `geva.input.filter` function can be used to remove rows whose p-values are all above a certain threshold (*e.g.*, 0.05), as demonstrated below:

```
# Removes the rows that are entirely composed by
# insignificant values
ginput <- geva.input.filter(ginput, p.value.cutoff = 0.05)
```

The correction above is applied using a threshold of $\alpha < 0.05$ for (corrected) p-values. Just like any other statistical procedure, the value of 0.05 given to the `p.value.cutoff` argument is arbitrary and it is upon the user's choice to define the best delimiter of significance.

4.3 Renaming the row names

Although large-scale experimental data is usually targeted to the context of each gene, it is particularly common in microarrays to use multiple probes that detect the expression levels for one or more genes. If one desires to use gene names as primary row identifiers instead of probes, these genes must replace the probes names accordingly. However, multiple genes per probe become duplicates, so one of them must chosen to provide unique identifiers for row names. In this sense, the `geva.input.rename.rows` function is used to perform the renaming while also solving such duplicates as demonstrated below:

```
# Replaces the row names with the "Symbol" column while
# selecting the most significant duplicates
ginput <- geva.input.rename.rows(ginput,
                                 attr.column = "Symbol")
```

In the example above, the `ginput` has an additional column called `"Symbol"` (accessed by `featureTable(ginput)$Symbol`) which is used to replace the row names, but the `attr.column` argument could also be a character vector with the same length of the number of rows. By default, the above code will select the duplicates which have the least p-values (*i.e.*, lowest error probability), which is also specified by applying the `dupl.rm.method="least.p.vals"` parameter. Alternatively, the parameter `dupl.rm.method="order"` can be used to select the duplicated value that appears first in the row order.

5 SV Analyses

By concluding the input step, a `GEVAInput` object that stores *logFC* values and weights becomes available in the current session. The next step will be to calculate the *summarization and variation* (SV) from the concatenated input data to produce the SV points, which are used in intermediate steps before the final classification.

5.1 Summarization

The `geva.summarize` function takes a `GEVAInput` object and performs the summarization, as demonstrated below:

```
# Summarizes ginput to find the SV points
gsummary <- geva.summarize(ginput)
```

The code above uses the default parameters for `summary.method` and `variation.method` (`"mean"` and `"sd"`, respectively) but other methods are available such as `"median"` and `"mad"` (median absolute deviation, or MAD). In this context, they could be specified as follow:

```
# Summarizes ginput using median and MAD
gsummary <- geva.summarize(ginput,
                           summary.method = "median",
                           variation.method = "mad")
```

In addition, all the summarization methods specified in `summary.method` and `variation.method` are implemented to take weights into account (except if not available or when weights are equivalent).

As a result, `geva.summarize` returns a `GEVASummary` object storing the table of `S` and `V` values. From this point, all objects defined by intermediate steps can be plotted as a *SV-plot*, a type of scatter plot where each point

(called *SV-point*) represents a gene's central *logFC* value (S) and *logFC* variation (V). For instance, a plot can be produced by calling the `plot` function on a `GEVASummary` object:

```
plot(gsummary)
```

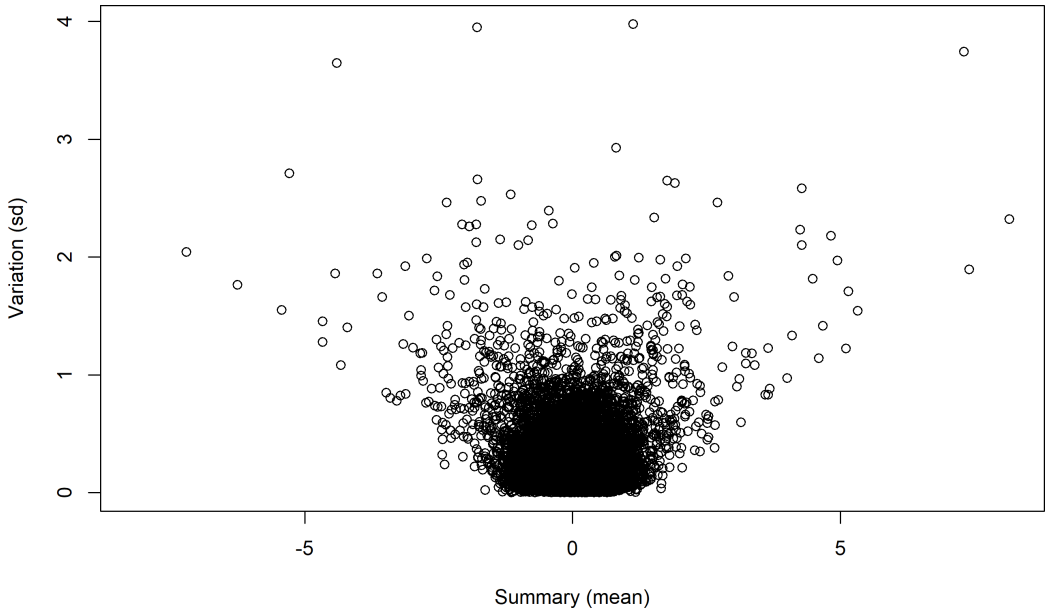

Figure 1: SV-plot produced from a `GEVASummary` object using the `geva.summarize` function with default parameters.

5.2 Delimitation by quantiles

After obtaining the `GEVASummary` object, the next step will be calculating the quantiles for every SV-point. That can be done by calling the `geva.quantiles` function as shown below:

```
# Calculates the quantiles from a GEVASummary object
gquants <- geva.quantiles(gsummary)
```

The code above produces a `GEVAQuantiles` object which stores the relevant partitions where the SV-points belong to. These partitions can be viewed by calling `plot` on the produced object:

```
plot(gquants)
```

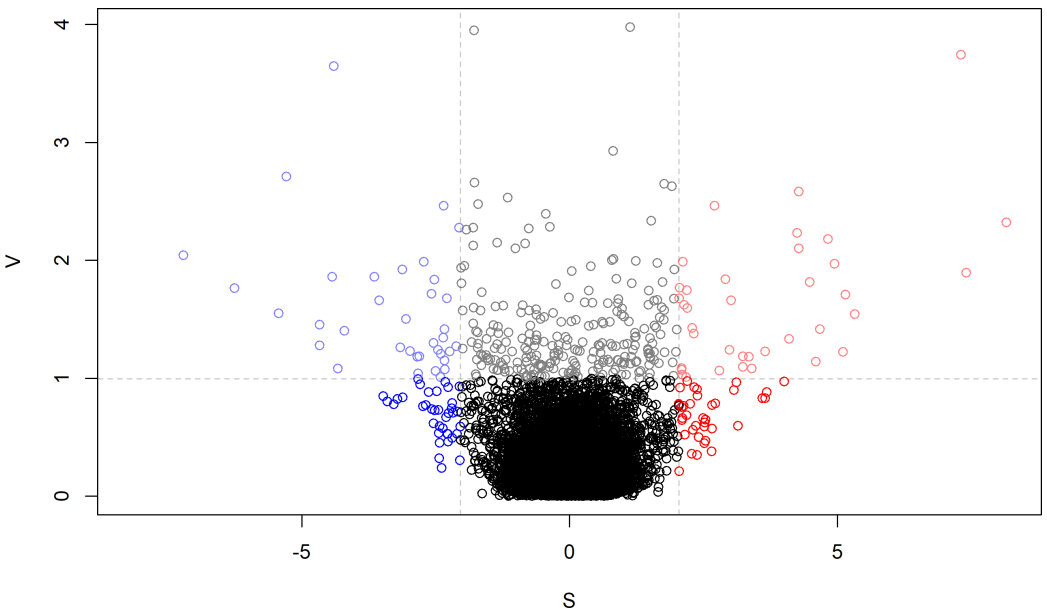

Figure 2: SV-plot produced from a `GEVAQuantiles` object using the `geva.quantiles` function with default parameters.

By default, the quantile detection is performed automatically using the parameter `quantile.method="range.slice"` (for more methods, call `?geva.quantiles`). However, the quantile delimiters can also be specified in the `initial.thresholds` argument like the following example:

```
# Calculates the quantiles from a GEVASummary object
# using custom delimiters
gquants <- geva.quantiles(gsummary,
                          initial.thresholds = c(S=1, V=0.5))
```

In this second example, thresholds of `1` and `0.5` were defined for `S` and `V` axes. As it can be noted from the SV-plot below, the results are purposely exaggerated and may not represent a good separation between relevant points, but this option is particularly useful to fine-tuning the quantile delimiters in situations where the automatic methods did not present a satisfatory outcome.

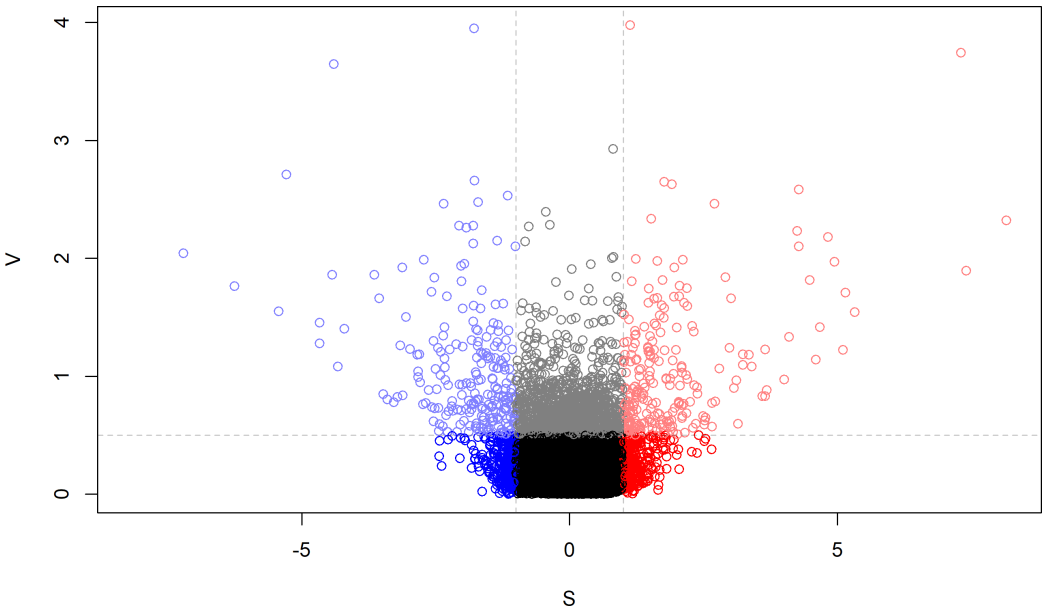

Figure 3: SV-plot produced from a `GEVAQuantiles` object using the `geva.quantiles` function using the `initial.thresholds = c(S=1, V=0.5)` parameter.

Note that the quantile detection does not define an absolute cutoff, but partitionizes the SV space into estimated regions containing qualitative classifications for the SV points. These classifications may change after combining the `GEVAQuantiles` with the results from the next steps.

5.3 Clustering

In this step, a cluster analysis is applied to separate relevant points from the agglomeration of non-differentially expressed genes. Such agglomeration

is mostly proeminent at the bottom-center region of a SV plot and essentially portraits the least relevant portion of the results.

5.3.1 Basic usage of the wrapper function

The `geva.cluster` function is the top-level function for clusters analysis and acts as a wrapper for more specific functions used to group SV points. The inner function is specified by the `cluster.method` argument with one of the following parameters: (i) `"hierarchical"`, calls the `geva.hcluster` function for hierarchical clustering; (ii) `"density"`, calls the `geva.dcluster` function for density-based clustering; and (iii) `"quantiles"`, calls the `geva.quantiles` function shown in the previous section. Likewise, optional parameters from the top funtion are passed to these calls.

In this section, only hierarchical and density-based clustering methods are going to be discussed. Both methods use the `resolution` argument, a single `numeric` between 0 and 1 that defines the ratio of output clusters. If the `resolution` is `0.0` (zero), the least number of clusters is assigned (*i.e.*, usually one or two), while if `1.0` then the maximum amount of clusters is assigned (*i.e.*, aproximately one cluster per point for hierarchical clustering). For example, to apply `geva.cluster` using hierarchical clustering at 30% of the resolution, the function is called as follows:

```
# Applies cluster analysis (30% resolution)
gcluster <- geva.cluster(gsummary,
                         cluster.method="hierarchical",
                         resolution=0.3)
```

The returned cluster data can be plotted using the generic `plot` function:

```
plot(gcluster)
```

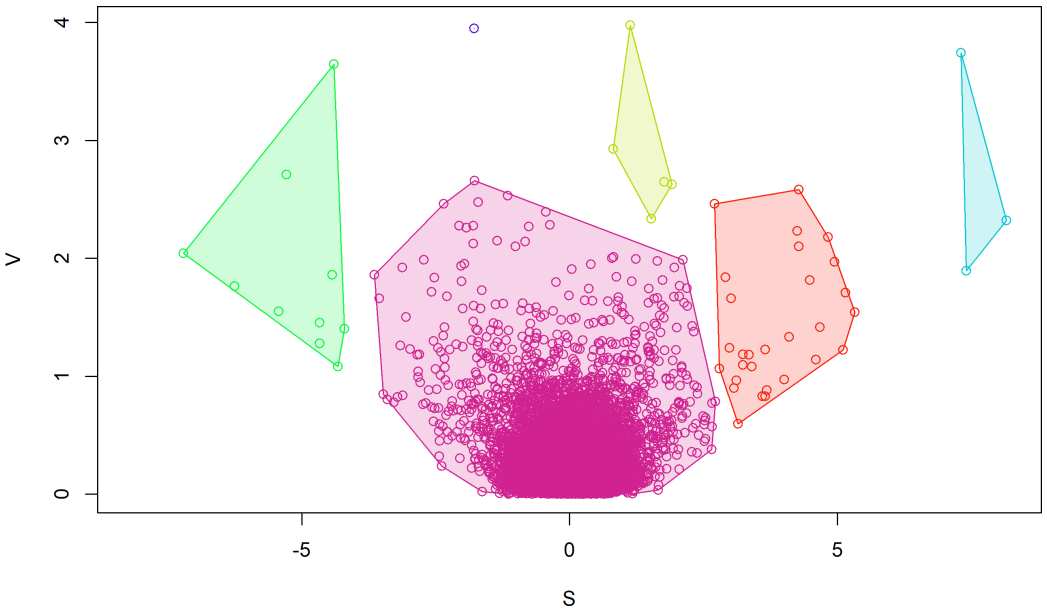

Figure 4: SV-plot produced from a `GEVACluster` object using the `geva.cluster` function with the hierarchical method and 30% of resolution.

5.3.2 Combining clusters with summarized data (Optional)

Apart from its usage as a wrapper, the `geva.cluster` function can also concatenate the summarized and grouped data into a single object by setting `grouped.return=TRUE` in the arguments. With this setup, the function will return a `GEVAGroupedSummary` object, which is a `GEVASummary` that includes the list of group sets (`GEVACluster` or `GEVAQuantile` objects). The code below illustrates this specific use case:

```
# Applies cluster analysis with default parameters and
# returns a GEVAGroupedSummary
ggroupedsummary <- geva.cluster(gsummary,
                                grouped.return = TRUE)
```

Alternatively, multiple group sets (clusters and quantiles) can be combined directly to the summarized data by appending each of

them with `groupsets<-`, which also converts the `GEVASummary` to a `GEVAGroupedSummary` object. For example, assuming that `gquants` and `gcluster` are output values from the previous quantiles (`geva.quantiles`) and clustering (`geva.cluster`) steps, respectively, the code would be:

```
# Makes a safe copy of the summary data
ggroupedsummary <- gsummary
# Appends the quantiles data
groupsets(ggroupedsummary) <- gquants
# Appends the clustered data
groupsets(ggroupedsummary) <- gcluster

# Draws a SV plot with grouped highlights (optional)
plot(ggroupedsummary)
```

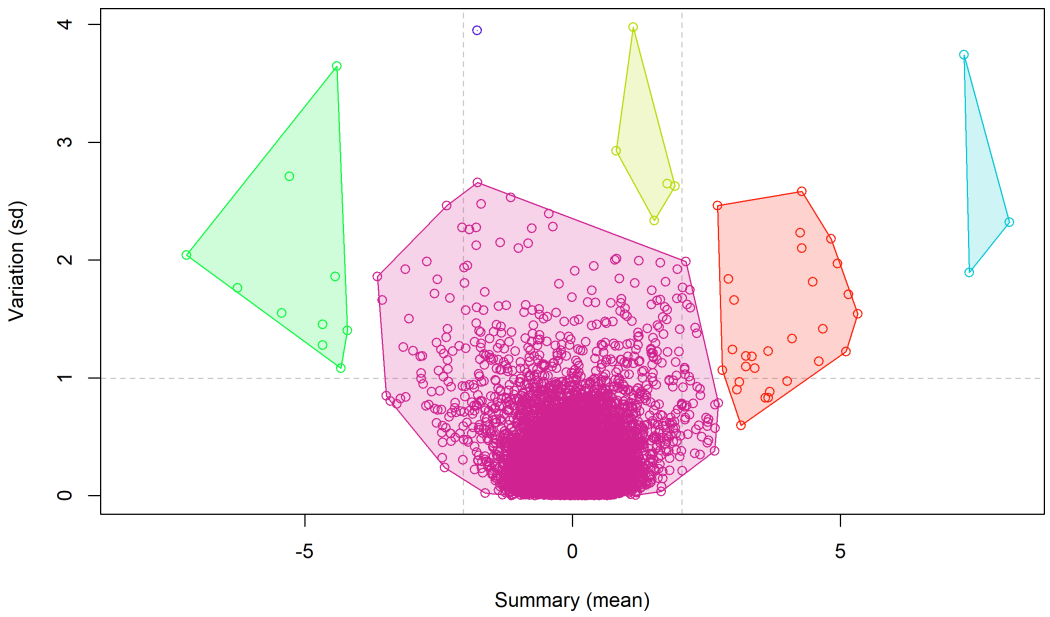

Figure 5: SV-plot produced from a `GEVAGroupedSummary` object after appending the `GEVAQuantiles` and `GEVACluster` objects from previous steps.

6 Attaining and accessing the results

After obtaining the quantiles and clusters from the summarized data in the previous step, now the entire data can be taken together to prospect the final classifications for each gene. This section presents the final steps to obtain the results table and some basic method to access it.

6.1 Final concatenation and factor analysis

In this final analysis step, the `geva.finalize` function takes a `GEVASummary` object as argument in addition to the other values returned from the intermediate steps, including the `GEVAQuantiles` and `GEVACluster` objects. Alternatively, a `GEVAGroupedSummary` object containing these intermediate results can be provided alone. The function will correct the quantiles based on the clustered points and return a classification that fits better both group assignments. Furthermore, if factors (groups of experimental conditions) were defined for the input columns, `geva.finalize` will also look for DE variations according to these factors, thereby unlocking two additional possible classifications (`"factor-dependent"` and `"factor-specific"`). The possible use cases are discussed in the following sub-sections:

6.1.1 Alternative 1 – Without factors

If factors were not included, no additional steps are required. The function call is done by passing the `GEVASummary`, `GEVAQuantiles` and `GEVACluster` from previous steps:

```
# Calculates the final classifications based on the
# intermediate results from previous steps
gresults <- geva.finalize(gsummary, gquants, gcluster)
```

Or, if a `GEVAGroupedSummary` object is provided:

```
# Calculates the final classifications based on the
# intermediate results from previous steps
gresults <- geva.finalize(ggroupedsummary)
```

Note that, without factors, the only relevant classification is `"similar"` (*i.e.*, genes with similar *logFC* values among all experiments).

6.1.2 Alternative 2 – With factors

Factors can be accessed and assigned to a `GEVAInput` object using `factors` and `factors<-`, respectively, and both accessors are valid for `GEVASummary` as well. The factors being set must be a `factor` or `character` vector whose length is equivalent to the number of columns, and it must contain at least two values per level to be considered since the factor analysis is based on ANOVA.

For instance, considering a `GEVASummary` object that stores a `GEVAInput` with 9 columns (experimental results), if one wants to separate these columns into 3 factors ('g1', 'g2', and 'g3'), the following code could be applied:

```
# Assigning factors to an example input with 9 columns

# Example with GEVAInput
factors(ginput) <- c('g1', 'g1', 'g1',
                     'g2', 'g2', 'g2',
                     'g3', 'g3', 'g3')

# Example with GEVAInput (using factor class)
factors(ginput) <- factor(c('g1', 'g1', 'g1',
                            'g2', 'g2', 'g2',
                            'g3', 'g3', 'g3'))
```

```
# Example with GEVASummary
factors(gsummary) <- c('g1', 'g1', 'g1',
                       'g2', 'g2', 'g2',
                       'g3', 'g3', 'g3')
```

By including factors in the current analysis, some optional arguments related to the factor analysis become available in `geva.finalize`. The `p.value`, for instance, determines the significance cutoff employed in ANOVA tests (by default, this value is `0.05` for $\alpha < 0.05$). In this case, the function call becomes:

```
# Calculates the final classifications based on the
# intermediate results from previous steps
gresults <- geva.finalize(gsummary, gquants, gcluster,
                          p.value=0.05)
```

Or, if a `GEVAGroupedSummary` object is provided:

```
# Calculates the final classifications based on the
# intermediate results from previous steps
gresults <- geva.finalize(ggroupedsummary, p.value=0.05)
```

The results can be plotted into a SV plot similarly as in the previous steps, but now only the relevant points will be colored while the rest are painted in black or gray:

```
plot(gresults)
```

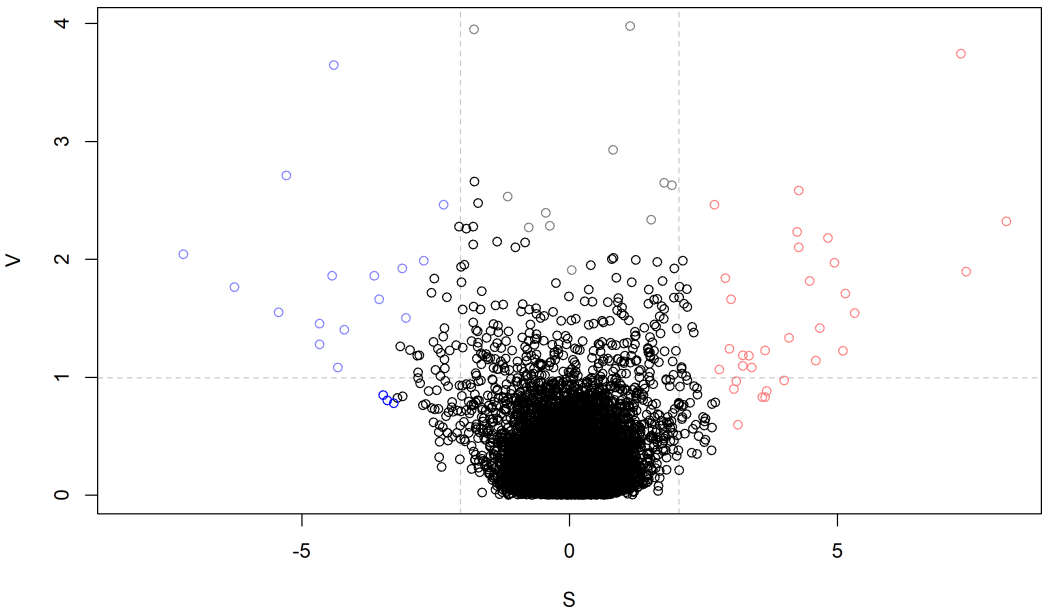

Figure 6: SV-plot produced from a `GEVAResults` object using the `geva.finalize` function with `0.05` as p-value cutoff.

6.2 Accessing and extracting the results

The returned `GEVAResults` object from `geva.finalize` represents the concatenation of all previous steps in addition to the results table and, if applicable, the intermediate steps from the factor analysis. The results table stores the final gene classifications, including the relevant (`"similar"`, `"factor-dependent"`, and `"factor-specific"`) and irrelevant (`"sparse"` and `"basal"`) ones. Each classification can be briefly described as follows:

- `basal`: Genes with similar but mild *logFC* that approximates to zero. Note that despite this name they not necessarily represent basal levels of gene expression, especially if the control group from DE analysis is not under normal conditions;
- `sparse`: Genes with high *logFC* variation but lacking any relationship to the experimental conditions or the factors;
- `similar`: Genes with relevant *logFC* (far from zero) and low *logFC* variance;
- `factor-dependent`: Genes with low *logFC* variance within the specified factors, but high variance between diferent factors;
- `factor-specific`: Genes with low *logFC* variance within one specific factor.

The function `results.table` can be used to return the table of final gene classifications:

```
tail(results.table(gresults), 10)
```

	classification	specific.factor
probe_9991	basal	NA
probe_9992	basal	NA
probe_9993	basal	NA

	classification	specific.factor
probe_9994	basal	NA
probe_9995	basal	NA
probe_9996	basal	NA
probe_9997	basal	NA
probe_9998	factor-specific	Cond_2
probe_9999	basal	NA
probe_10000	basal	NA

On the other hand, the `top.genes` function may be a rather practical way to return the most relevant results. It extracts by default the `"similar"`, `"factor-dependent"`, and `"factor-specific"` results, and can attach additional columns (*e.g.*, gene symbols) specified by the `add.cols` arguments. The code below shows an usage example of `top.genes`:

```
# Extracts the top genes only
dtgens <- top.genes(gresults)

# Extracts the top genes and appends the "Symbol" column
dtgens <- top.genes(gresults, add.cols = "Symbol")

# Prints the last lines of the top genes table (optional)
print(tail(dtgens, 10))
```

	Symbol	classification	specific.factor
probe_8487	GENE_K8487	factor-dependent	NA
probe_8740	GENE_D8740	factor-dependent	NA
probe_8823	GENE_I8823	factor-specific	Cond_1
probe_9136	GENE_J9136	similar	NA

	Symbol	classification	specific.factor
probe_9312	GENE_D9312	factor-dependent	NA
probe_9495	GENE_E9495	factor-dependent	NA
probe_9601	GENE_G9601	factor-specific	Cond_3
probe_9758	GENE_H9758	factor-specific	Cond_3
probe_9893	GENE_M9893	factor-dependent	NA
probe_9998	GENE_N9998	factor-specific	Cond_2

The resulting table can then be exported using functions such has `write.table` from the R base package.

6.3 Shortcut function and reanalysis

The `geva.quick` function accepts a `GEVAInput` object and performs all intermediate functions from the summarization to the final concatenation. Optional (...) arguments are passed to the internal calls to `geva.summarize`, `geva.quantiles`, `geva.cluster` and `geva.finalize`, ultimately returning a `GEVAResults` object. The basic usage is described as follows:

```
# Generates a random GEVAInput example
ginput <- geva.ideal.example()
# Performs all intermediate steps with geva.quick
# The resolution is used by the call to geva.cluster
gresults <- geva.quick(ginput, resolution=0.25)
## > Found 4 clusters and 31 significant genes
gresults <- geva.quick(ginput, resolution=0.4)
## > Found 16 clusters and 116 significant genes
```

This function can be applied to a `GEVAResults` object as well to restore the parameters that produced this result, whereas optional (...) arguments can overwrite them:

```r
# Generates a random GEVAInput example
ginput <- geva.ideal.example()
# Performs all intermediate steps with geva.quick
# The summary.method is used by the call to geva.summarize
gresults <- geva.quick(ginput, summary.method='mean')
## > Found 60 significant genes
gresults <- geva.quick(gresults, summary.method='median')
## > Found 95 significant genes
```

In the example above, the entire analysis was redone using the overwritten `summary.method` argument. Therefore, by following this pattern, users can tweak different parameters depending on their statistical choice regarding the current biological context.

References

[1] Gordon K Smyth. "Limma: linear models for microarray data". In: *Bioinformatics and computational biology solutions using R and Bioconductor*. Springer, 2005, pp. 397–420.

[2] Michael I Love, Wolfgang Huber, and Simon Anders. "Moderated estimation of fold change and dispersion for RNA-seq data with DESeq2". *Genome biology* 15.12 (2014), pp. 1–21.

[3] Itamar J G Nunes. "Gene expression analysis platform (GEAP): uma plataforma flexível e intuitiva para análise de transcriptoma". *LUME UFRGS* (2018).

[4] M Aleksi Kallio et al. "Chipster: user-friendly analysis software for microarray and other high-throughput data". *BMC genomics* 12.1 (2011), pp. 1–14.