CAFE Manual

Sander Bollen

October 17, 2016

Contents

1 Introduction

So, you have downloaded the **CAFE** package, and are wondering "What on Earth can I do with this?". In answering this question, we can first of all refer to the package name: I'm sure you'd all have thought about a nice warm cup of coffee, but in reality **CAFE** stands for **C***hromosomal* **A***berrations* **F***inder in* **E***xpression data*. Now, that might not tell you much, so here is a slightly better - albeit longer - summary of what it does: **CAFE** analyzes microarray expression data, and tries to find out whether your samples have any gross chromosomal gains or losses. On top of that, it provides some plotting functions - using the *ggplot2* package - to give you a nice visual tool to determine where those aberrations are located. How it exactly does this is something you can find later in this document. CAFE does not invent any new algorithms for the detection of chromosomal aberrations. Instead, it takes the approach of Ben-David *et al*, and molds it into an easy-to-use R package.

1.1 Prerequisites

As all software packages, **CAFE** has its requirements. You should preferably have the latest version of R, but at the very least version 2.10. Other dependencies can be found in the DESCRIPTION file. Some packages imported by **CAFE** require you to have `libcurl` installed in your system, and by extension the *RCurl* and *Rtracklayer* packages. If these are not yet installed in your system, this could take a while to install.

1.2 Preparing your file system

CAFE analyzes microarray expression data. That means that without anything *to* analyze, it won't do anything at all. So, how do we fix this obvious problem? As a starter, **CAFE** only analyzes .CEL files. This means that **CAFE** will only work with data from Affymetrix microarrays. Illumina or other platforms unfortunately will not work. To prepare your filesystem correctly, follow the following steps.

Put all CEL files in a folder and then start R in that folder. Alternatively, you can set the working directory to this folder by `setwd("/some/folder/path/")`. And we're done. It's that simple. See figure 1 for a screenshot of how this looks in a file manager.

See figure 1 for a screenshot of how this is layed out

2 Analysis

Now we're ready to start analyzing our dataset(s).

CAFE provides the `ProcessCels()` function. This function takes four arguments: `threshold.over`, `threshold.under` `remove_method` and `local_file`. The threshold will determine which probes are going to be considered as overexpressed. The default setting is `threshold.over=1.5`, meaning that probes with a relative expression over 1.5 times median will be considered overexpressed. Likewise `threshold.under` determines which probesets will be considered underexpressed. As for the `remove_method` argument, this determines in what way `ProcessCels` will remove duplicate probes. See section 3.1 for an overview of this option. The `ProcessCels()` function will basically suck up all your CEL files in your working directory - but don't worry, they'll still be there in your file system - and perform some number magic on them; it will normalize, calculate probes that are overexpressed, log2 transform and remove duplicates. It returns you a `list` object of your entire dataset, which the rest of **CAFE** requires to function.

So lets try that. First of all, we of course need to load the package:

```
> library(CAFE)
```

Then, we should set our working directory to the dataset folder if we're not already there.

```
> setwd("~/some/path")
```

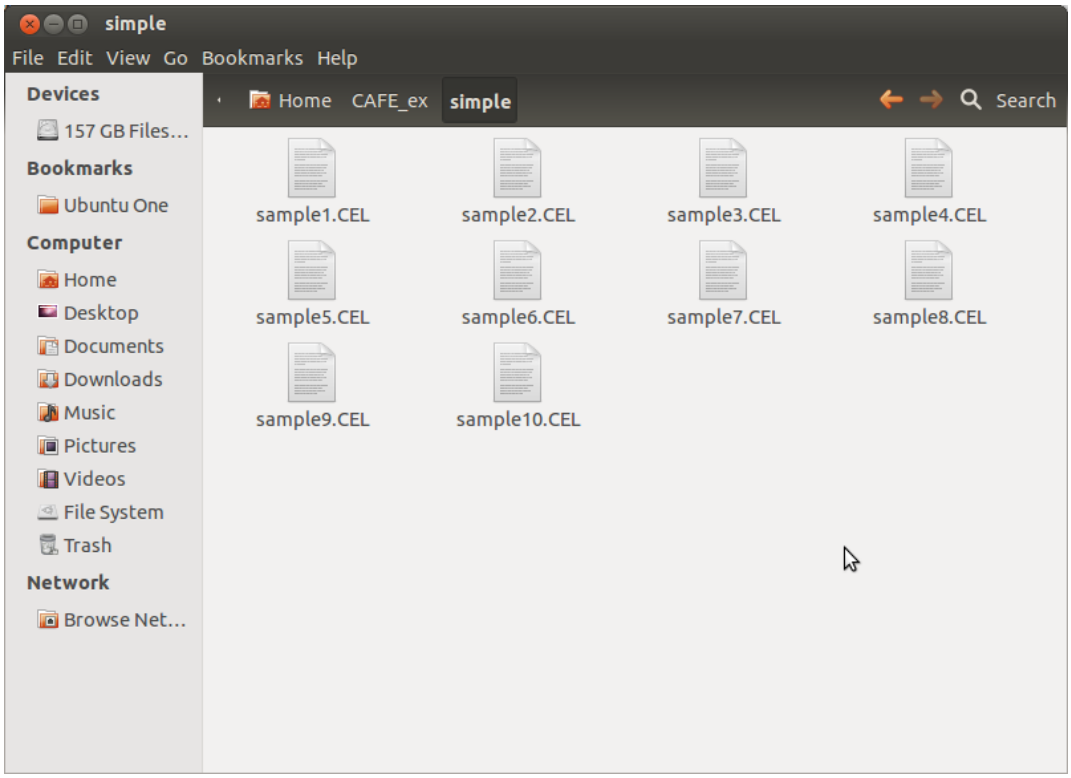

Figure 1: The file system layout

Now we're there we can process the CEL files.

```
> datalist <- ProcessCels()
```

2.1 A moment of reproducibillity

The stuff we've done above here is unfortunately not really reproducible. It requires you to have .CEL files in a folder called `/some/path` in your home directory, and that's of course not very pretty. So to keep the rest of this document reproducible, the **CAFE** package comes together with a data object. This is the `list` object returned by `ProcessCels()` when processing both GSE10809 and GSE6561.

```
> data("CAFE_data")
> #will put object named {\bf CAFE}_data in your global environment
```

2.2 Statistics

So now we come to the core of **CAFE**: finding which chromosomes are significantly over- or underexpressed. **CAFE** uses *thresholding* to determine which probes are overexpressed or underexpressed. We use the probes we deemed overexpressed by our threshold to create a contingency table which can then be used in an Exact Fisher or Chi-Square test. Basically we assume that everything over our defined threshold.over is really overexpressed, and everything uder our defined threshold.under is underexpressed.

To calculate p-values for chromosomes, we will use the `chromosomeStats()` function. We can then also select which test (fisher or chi square) we want to use. When performing multiple tests (i.e. if we are testing multiple chromosomes), we need to correct our p-values for type I errors. We can therefore set the `bonferroni` argument to true when testing multiple chromosomes. The `enrichment` argument controls which side (over- or underexpressed) we are testing for.

```
> #we first have to decide which samples we want to use.
> names(CAFE_data[[2]]) #to see which samples we got

 [1] "GSM151738.CEL" "GSM151739.CEL" "GSM151740.CEL" "GSM151741.CEL" "GSM272914.CEL"
 [6] "GSM272915.CEL" "GSM272916.CEL" "GSM272917.CEL" "GSM272918.CEL" "GSM272919.CEL"
[11] "GSM272920.CEL" "GSM272921.CEL"

> sam <- c(1,3) #so we use sample numbers 1, and 3 to compare against the rest
> chromosomeStats(CAFE_data,chromNum=17,samples=sam,
+                 test="fisher",bonferroni=FALSE,enrichment="greater")

      Chr17
3.989436e-48

> # we are only testing 1 chromosome
```

This uses an Exact Fisher test. Technically speaking, this is better than a chi square test, but can be slower for very large sample sizes. We can also do a chi square test, which is slightly faster.

```
> chromosomeStats(CAFE_data,chromNum=17,samples=sam,
+                 test="chisqr",bonferroni=FALSE,enrichment="greater")

      Chr17
1.187963e-05
```

As you will have seen, the output of this is different from `test="fisher"`. The reason for this is that the Fisher test gives an *exact* p-value, whereas a chi square test is just an approximation.

But if we now want to test multiple chromosomes, we have to correct our p-values

```
> chromosomeStats(CAFE_data,chromNum="ALL",samples=sam,
+                 test="fisher",bonferroni=TRUE,enrichment="greater")

       Chr1         Chr2         Chr3         Chr4         Chr5         Chr6
1.000000e+00 1.000000e+00 1.000000e+00 1.020544e-01 3.636588e-01 1.000000e+00
       Chr7         Chr8         Chr9        Chr10        Chr11        Chr12
1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 3.808581e-42
      Chr13        Chr14        Chr15        Chr16        Chr17        Chr18
3.868285e-01 1.000000e+00 1.000000e+00 1.000000e+00 8.776760e-47 1.000000e+00
      Chr19        Chr20        Chr21        Chr22
1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00
```

The results which we got might be all nice and well - there seems be something amiss with chromosome 17 - but preferably we would like to delve a bit deeper. We would like to know which chromosome *bands* are duplicated or lost. To do this we can use the same syntax as for chromosomes, except that we are using a different function:

```
> bandStats(CAFE_data,chromNum=17,samples=sam,test="fisher",
+           bonferroni=TRUE,enrichment="greater")

    17cen-q12       17p11.1       17p11.2   17p11.2-p12         17p12   17p12-p11.2
 1.000000e+00  1.000000e+00  8.008655e-17  1.000000e+00  1.000000e+00  1.000000e+00
     17p12.3         17p13     17p13-p12       17p13.1   17p13.1-p12       17p13.2
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
     17p13.3    17pter-p11           17q         17q11     17q11-q12     17q11-q21
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
  17q11-q21.3    17q11-qter       17q11.1 17q11.1-q11.2       17q11.2   17q11.2-q12
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
       17q12     17q12-q21   17q12-q21.1         17q21     17q21-q22     17q21-q23
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
    17q21-q24    17q21-qter       17q21.1 17q21.1-q21.3       17q21.2       17q21.3
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
```

```
  17q21.3-q22      17q21.31      17q21.32      17q21.33         17q22     17q22-q23
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
  17q22-q23.2       17q22.2         17q23     17q23-q24       17q23.1       17q23.2
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
17q23.2-q25.3       17q23.3         17q24     17q24-q25       17q24.1       17q24.2
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
      17q24.3         17q25       17q25.1       17q25.2 17q25.2-q25.3       17q25.3
 1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00  1.000000e+00
       17qter
 1.000000e+00
```

```
> #multiple bands per chromosome, so need bonferroni!
```

So this way we see that there is most likely a duplication around band 17p11.2

2.3 Plotting

So now we know that chromosome 17 for these two samples is aberrant, but we would like to plot that. A picture says more than a thousand words - as the saying goes. **CAFE** provides four functions for plotting samples, `rawPlot()`, `slidPlot()`, `discontPlot()` and `facetPlot()`.

2.3.1 Raw plots

The `rawPlot()` function plots each individual probe along the chromosome with its 'raw', unaltered, log2 relative expression value. This can give a very rough overview of what is happening. As a visual tool, an ideogram of the chromosome can be plotted over the plot. See figure 2 for an example.

```
> p <- rawPlot(CAFE_data,samples=c(1,3,10),chromNum=17)
```

```
> print(p)
```

2.3.2 Sliding plots

The plots given by `rawPlot()` are often not very informative since the within-sample variation in expression levels is quite high. The `slidPlot()` function solves this problem by applying a sliding average to the entire sample. As such, patterns become visible that would otherwise have went unnoticed. The function has two extra arguments as `rawPlot()`: if `combine=TRUE` a raw plot will be plotted in the background. The size of the sliding window can be determined by argument `k`. See figure 3 for an example.

```
> p <- slidPlot(CAFE_data,samples=c(1,3,10),chromNum=17,k=100)
```

```
> print(p)
```

2.3.3 Discontinuous plots

In reality, there can only be an integer number of copy numbers for a given region (be it an entire chromosome or a band). One cannot 2.5 times duplicate a region; no, it is 0, 1, 2, 3 times etc. Also, there should be a defined boundary where the duplication or loss begins and ends. Yet, functions like `slidPlot()` will give smooth transitions and variable regions, which is a relatively poor reflection of what is actually happening. As such, we need a discontinuous smoother rather than a sliding average smoother. A discontinuous smoother is a smoother which produces distinct "jumps" rather than smooth transitions. The `discontPlot()` function implements such a discontinuous smoother - called a *Potts filter*. The smoothness (i.e. amount of jumps) can be determined by setting parameter `gamma`. A higher gamma will result in a smoother graph, with less jumps. See figure 4 for an example.

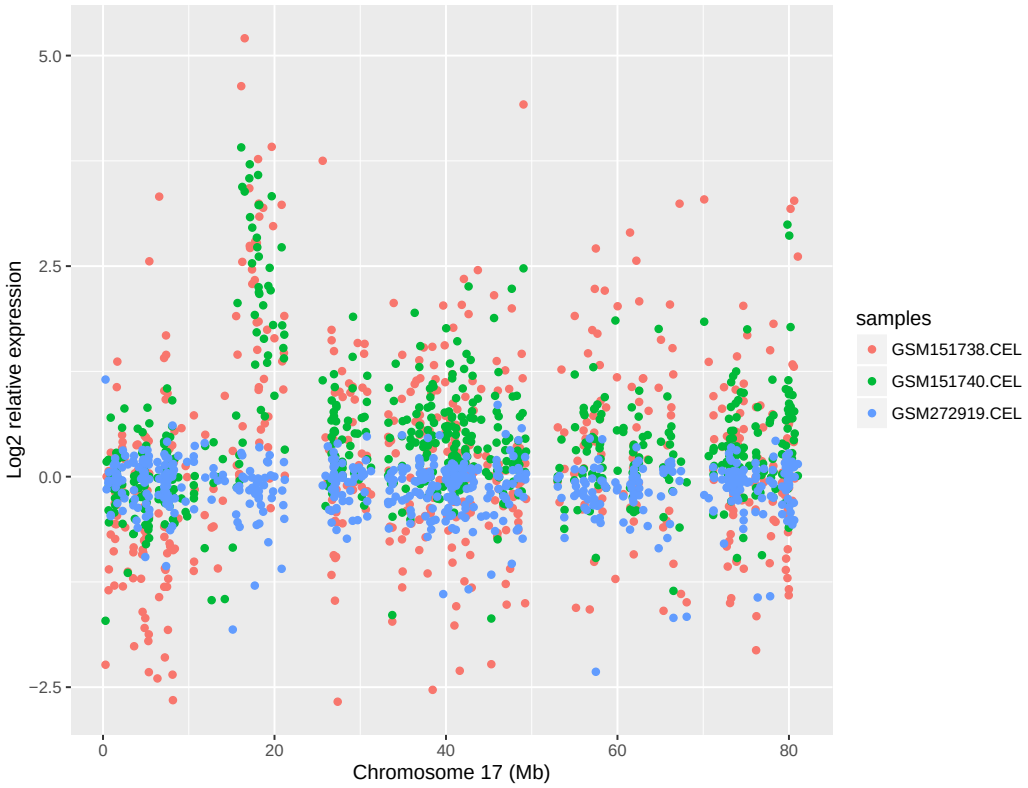

Figure 2: A rawPlot of samples 1, 3 and 10 for chromosome 17

```
> p <- discontPlot(CAFE_data,samples=c(1,3,10),chromNum=17,gamma=100)
> print(p)
```

2.3.4 Facet plots

With the `facetPlot()` function you can plot all chromosomes stitched together horizontally in one single splot. It includes options to use either a sliding average smoother. See figure 4 for an example..

```
> p <- facetPlot(CAFE_data,samples=c(1,3,10),slid=TRUE,combine=TRUE,k=100)
> print(p)
```

3 In a nutshell

As follows from the above, CAFE analysis basically boils down to just three steps

1. `data <- ProcessCels()`
2. `xxxStats(data, ...)`
3. `xxxPlot(data, ...)`

For instance:

```
> data <- ProcessCels()
> chromosomeStats(data,samples=c(1,3),chromNum="ALL")
> discontPlot(data,samples=c(1,3),chromNum="ALL",gamma=100)
```

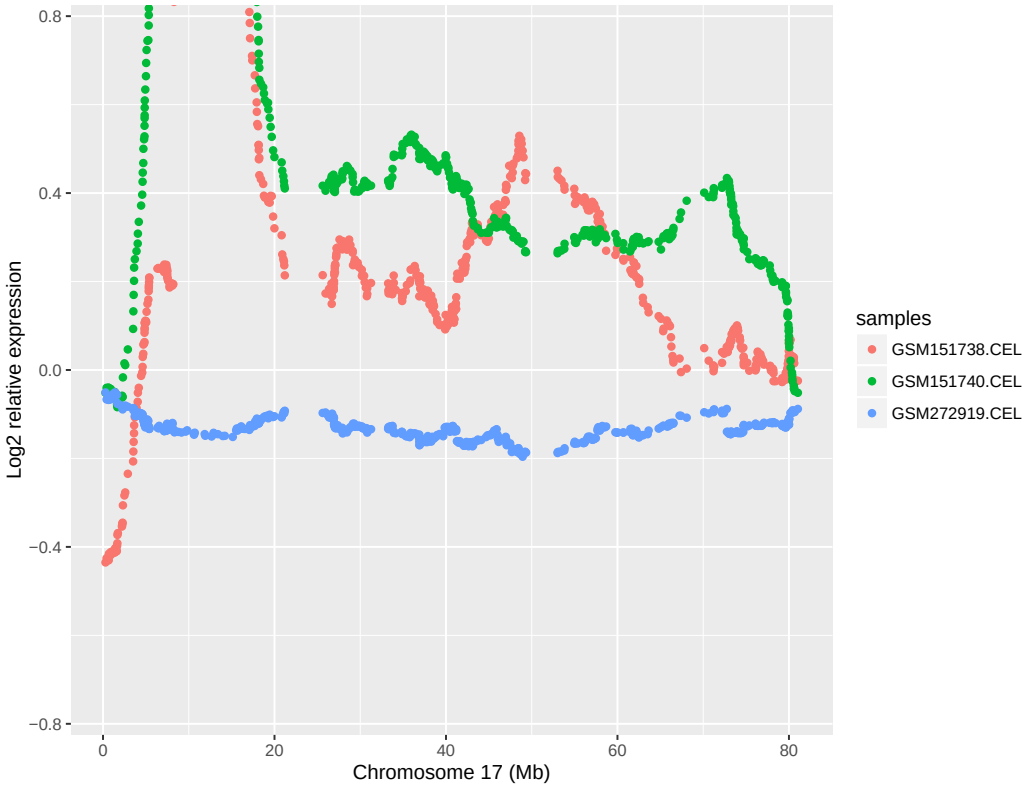

Figure 3: A slidPlot of samples 1, 3 and 10 for chromosome 17 with a sliding window size of 50, with raw data in the background

4 Behind the scenes

4.1 Normalization

Affymetrix CEL files are read in and normalized by the `justRMA()` function in the *affy* package. Absent probes are then found (by the `mas5calls` function), and are subsequently removed if absent in more than 20% of samples. After normalization is complete, relative expression values (to median) are calculated. The `remove_method` argument controls which duplicate probes are removed from the dataset. When `remove_method=0`, no duplicate probes will be removed. When `remove_method=1`, duplicate probes linking to the same gene will be removed such that each gene only links to one single probe, according to the following priorities:

1. Probes with `..._at` are preferably retained
2. Probes with `...a_at`
3. Probes with `...n_at`
4. Probes with `...s_at`
5. And finally probes with `...x_at`

When this still results in multiple probes per gene, the probe with the earliest chromosomal location is retained. When `remove_method=2`, duplicate probes are only removed when they link to the exact same location, using the same priority list as specified above.

4.2 Category testing

When using `chromosomeStats()` and `bandStats()` the dataset is eventually split into four categories:

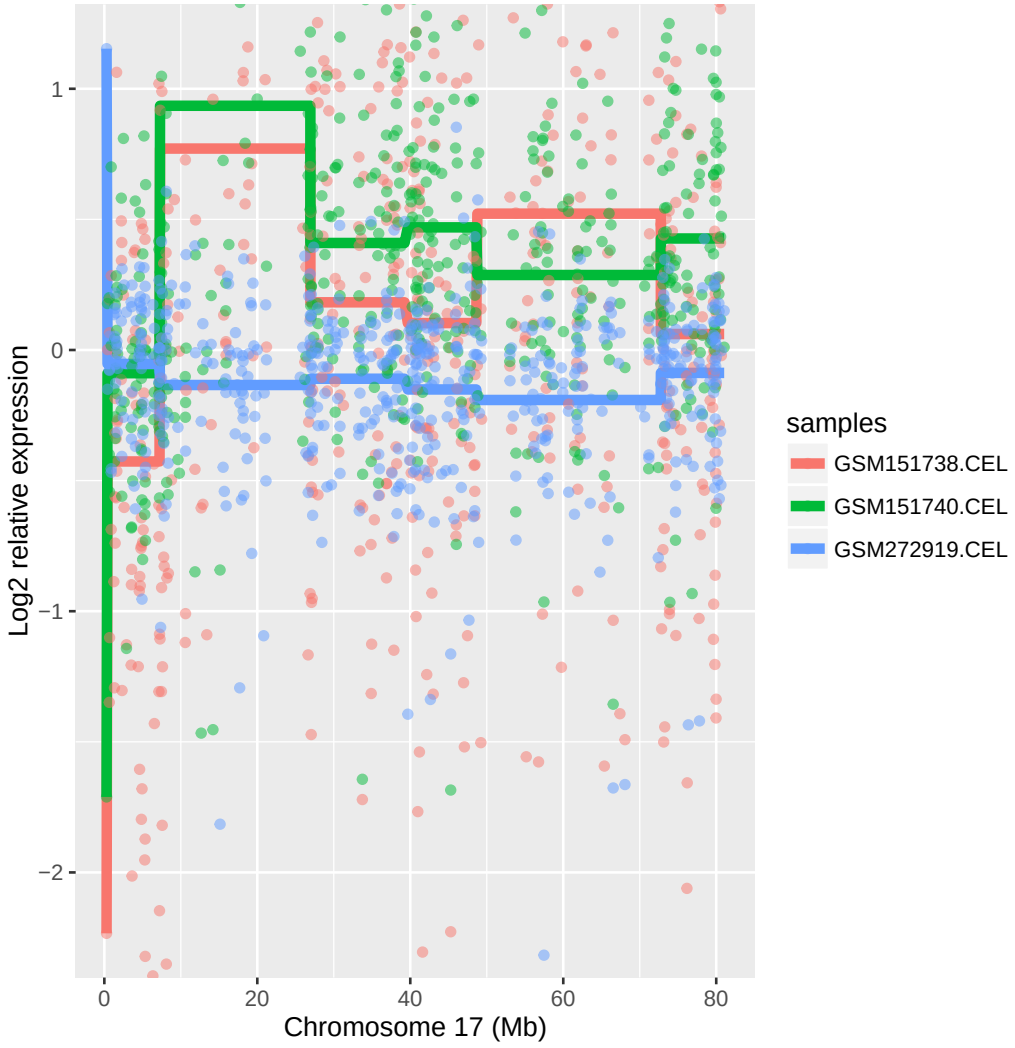

Figure 4: A discontPlot of samples 1, 3 and 10 for chromosome 17 with a gamma of 50, with raw data in the background

Figure 5: A facetPlot of samples 1, 3 and 10 with a sliding average with window size of 50, with raw data in the background

1. Whole dataset & On Chromosome (or band)
2. Whole dataset & Overexpressed & On Chromosome (or band)
3. Sample(s) & On Chromosome (or band)
4. Sample(s) & Overexpressed & On Chromosome (or band)

The number of probes for each category are counted, and saved in a so-called contingency table. Then we test the likelihood that Sample(s) occurs within Whole via either the Exact Fisher test or the Chi Square test. The Exact Fisher test supplies *exact* p-values, whereas the Chi Square test supplies merely an approximation. However, the Exact Fisher test can be slower than a Chi Square test, and the Chi Square test increases in accuracy with increasing sample size.

"Overexpressed" is defined as those probes which were over the set threshold - the `threshold.over` argument in `ProcessCels()`. Likewise, "underexpressed" is defined as those probes which were under the set threshold - argument `threshold.under`. Please note that these thresholds are a defined as a fraction of median expression value for each probe.

4.3 Smoothers

4.3.1 Sliding average

With the moving average we attempt to the smooth the raw data so that patterns emerge. We have a set of points $(x_i, y_i), i = 1 \ldots n$, which is ordered such that $x_1 \leq \cdots \leq x_n$. Since we plotting chromosomal location versus log2 relative expression, x_i refers to chromosomal locations and y_i refers to log2 relative expressions. Since chromosomal locations are integers, $x_i \in \mathbb{N} : x_i \geq 0$, and since log2 relative expressions can be any real number $y_i \in \mathbb{R}$. We have a sliding average *windows*, denoted k, which determines the smoothness. We are reconstructiong y_i (the log2 relative expressions) to come to a smoother $\hat{y}_i$. For y_i at least k-elements from the boundary, the smoother is calculated as in equation 1. For the remaining $\hat{y}_i$, where $i \in \mathbb{N} : 1 \leq i \leq k$ and $i \in \mathbb{N} : (n-k+1) \leq i \leq n$, we first define two variables a and b, where $a = max(1, (i-k))$ and $b = min(n, (i+k))$. The remaining $\hat{y}_i$ are then calculated using equation 2.

$$\hat{y}_i = \frac{1}{k} \cdot \sum_{i=j}^{i+k} y_j, i \in \mathbb{N} : (k+1) \leq i \leq (n-k) \tag{1}$$

$$\hat{y}_i = \frac{\sum_{i=a}^{b} y_i}{(a-b+1)} \tag{2}$$

4.3.2 Discontinuous smoother

The discontinuous smoother used in `discontPlot()`, is essentially a minimization problem. We again have a set of points $(x_i, y_i), i = 1 \ldots n$. This set is ordered in such manner that $x_1 \leq \cdots \leq x_n$. For our particular problem x_i refers to chromosomal coordinates, such that $x_i \in \mathbb{N} : x_i \geq 0$. Conversely, y_i refers to log2 relative expressions, such that $y_i \in \mathbb{R}$. We want to reconstruct y_i in such a way that the reconstruction - called $\hat{y}_i$ - closely resembles y_i but has as little jumps as possible. A jump is defined as $\hat{y}_i \neq \hat{y}_{i+1}$. The reconstruction $\hat{y}_i$ can be constructed by minimizing the following function:

$$H = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 + \gamma \cdot |(\hat{y}_i \neq \hat{y}_{i+1})| \tag{3}$$

In the above formula, $||$ denotes cardinality. The first term in the formula denotes a least squares goodness-of-fit term. The second term determines how strict the formula is. A higher γ will result in a flatter result, with fewer jumps, and this γ can be any number over 0. The entire formula is called a *Potts filter*. The mathematics behind this formula is described in detail by Winkler et al.

The implementation in `discontPlot()` uses the algorithm descibed by Friedrich et al.

4.4 Non-Affymetrix data

Even though **CAFE** is designed to work only with Affymetrix .CEl files, it is possible to analyse non-affymetrix data via a workaround. For **CAFE** to work with non-affymetrix data, it is necessary you provide the correct data structure which we use to analyse our data. The major structure the software works with is a **list of lists** structure. The list contains **three** lists - named `$whole`, `$over` and `$under` - which both contain data frames for each and every sample using the following format:

1. `$ID`, some identifier (probe IDs in affymetrix) (character vector)
2. `$Sym`, gene symbol (character vector)
3. `$Value`, log2 transformed expression value (numerical vector)
4. `$LogRel`, log2 transformed relative expressions (as a function of mean) (numerical vector)
5. `$Loc`, chromosomal locations in bp (integer vector)
6. `$Chr`, chromosome number (character vector)
7. `$Band`, cytoband (character vector)

For instance

```
> str(CAFE_data$whole[[1]])
```

```
'data.frame':        10442 obs. of  8 variables:
 $ ID    : chr  "1053_at" "121_at" "1316_at" "1487_at" ...
 $ Sym   : chr  "RFC2" "PAX8" "THRA" "ESRRA" ...
 $ Value : num  6.48 6.72 4.17 6.39 7.41 ...
 $ LogRel: num  -2.077 -0.336 -0.435 -0.814 -1.672 ...
 $ Loc   : int  73646002 113974938 38219184 64073043 43562239 26722694 90039000 196439228 27941782 3417521
 $ Chr   : chr  "7" "2" "17" "11" ...
 $ Band  : chr  "7q11.23" "2q13" "17q11.2" "11q13" ...
 $ Arm   : chr  "7q" "2q" "17q" "11q" ...
 - attr(*, "na.action")=Class 'omit'  Named int [1:95] 44 53 54 55 56 57 187 194 206 220 ...
  .. ..- attr(*, "names")= chr [1:95] "44" "53" "54" "55" ...
```

The `$whole` list should contain data for all probes, the `$over` list should only contain those probes which are deemed overexpressed (not required if one uses `thresholding=FALSE`). The order of probes and locations in `$whole` should be *identical* acros s all samples. Each list element (i.e. the dataframes) is named according to its sample name. For instance

```
> print(names(CAFE_data$whole))
```

```
 [1] "GSM151738.CEL" "GSM151739.CEL" "GSM151740.CEL" "GSM151741.CEL" "GSM272914.CEL"
 [6] "GSM272915.CEL" "GSM272916.CEL" "GSM272917.CEL" "GSM272918.CEL" "GSM272919.CEL"
[11] "GSM272920.CEL" "GSM272921.CEL"
```

5 References

1. Uri Ben-David, Yoav Mayshar, and Nissim Benvenisty. Virtual karyotyping of pluripotent stem cells on the basis of their global gene expression profiles. *Nature protocols*, **8(5):989-997, 2013**.
2. G Winkler, Volkmar Liebscher, and V Aurich. Smoothers for Discontinuous Signals. *Journal of Nonparametric Statistics*, **14:203-222, 2002**.
3. F Friedrich, a Kempe, V Liebscher, and G Winkler. Complexity Penalized M- Estimation. *Journal of Computational and Graphical Statistics*, **17(1):201-224, March 2008**.