Getting Started with mutSignatures

Damiano Fantini

April 3, 2018

The mutSignatures Framework: extracting mutational signatures from collections of DNA mutations

Cancer cells accumulate DNA mutations as result of DNA damage and inaccurate DNA repair. *mutSignatures* is a computational framework aimed at deciphering DNA mutational signatures linked to genetic instability processes operating in cancer. The framework provides tools for importing DNA mutation counts, retrieve the *n*-nucleotide context surrounding each DNA mutation, and then perform non-negative matrix factorization to extract mutational signatures. Also, tools for deconvoluting catalogs of mutations against known mutational signatures (such as the COSMIC ones), and for data visualization are included. The framework can take advantage of parallelization and is already optimized for multi-core systems. This framework is an R-based implementation based on the MATLAB WTSI framework by Alexandrov LB et al (2013) doi:10.1016/j.celrep.2012.12.008, and comes with a series of improvements described by Fantini D et al (2018) doi:10.1038/s41388-017-0099-6.

Modules Included in the mutSignatures Framework

The *mutSignatures* pipeline is organized in three Modules:

- Module 1: Data Import and Preparation from VCF files, TCGA datasets, and other sources
- Module 2: *de novo* extraction of mutational signatures via NMF; alternatively, deconvolution of Mutational Catalogs using already known signatures
- Module 3: Result analysis and visualization

This vignette covers each module, and provides information and examples to install and use the latest version (1.3.1) of *mutSignatures*, which is available on GitHub.

Package Installation

The latest version of *mutSignatures* (version 1.3.1) is currently available on GitHub, at the following URL: https://github.com/dami82/mutSignatures. It is possible to install it using the *devtools* library.

```
# install devtools if needed
if(!"devtools" %in% rownames(installed.packages()))
  install.packages("devtools")

# install mutSignatures
library(devtools)
install_github("dami82/mutSignatures")

# load required libraries for this vignette
library(ggplot2)
library(mutSignatures)
```

Quick Start

The extraction of novel mutational signatures requires three steps:

1. prepare the mutation count matrix: this is achieved by importing a list of mutations, retrieving their *n*-nucleotide context, and then count mutation types in each sample/pateint;
2. perform non-negative matrix factorization (NMF) to extract mutation signatures
3. export and visualize the results

The *mutSignatures* framework can execute these operations with few lines of code. While each module will be covered in detail in the following sections, here you'll find a simple example to get started.

VCF files are download from an online repository, and then imported using the `importVCFfiles()` function. Mutation types are computed using the `processVCFdata()` function, and then counted for each of the input samples. This last operation is performed by the `countMutTypes()` function.

```
# Obtain the data: download a collection of VCF files from mouse tumors
vcf_url <- "http://www.labwizards.com/rlib/test_muts.tar.gz"
download.file(vcf_url, destfile="test_muts.tar.gz")
untar(tarfile = "test_muts.tar.gz", compressed = "gzip", list = F)
vcfFiles <- grep("^0[01][0-9]\\.vcf$", dir(), value = TRUE)

# Import and prepare data from multiple VCF files
allvcfData <- importVCFfiles(vcfFiles = vcfFiles)

# Load the mm10 genome and then compute mutation types
mm10 <- BSgenome.Mmusculus.UCSC.mm10::BSgenome.Mmusculus.UCSC.mm10
fullData <- processVCFdata(vcfData = allvcfData, BSGenomeDb = mm10)

# Count mutation types in each patient/sample
allVCF.counts <- countMutTypes(mutTable = fullData,
                               mutType_colName = "mutType",
                               sample_colName = "SAMPLEID")

# The result is a mutationCounts object
class(allVCF.counts)
## [1] "mutationCounts"

allVCF.counts
##  Mutation Counts object - mutSignatures
##
##  Total num of MutTypes: 96
##  MutTypes: A[C>A]A, A[C>A]C, A[C>A]G, A[C>A]T, A[C>G]A ...
##
##  Total num of Samples: 10
##  Sample Names: 001, 002, 003, 004, 005 ...
```

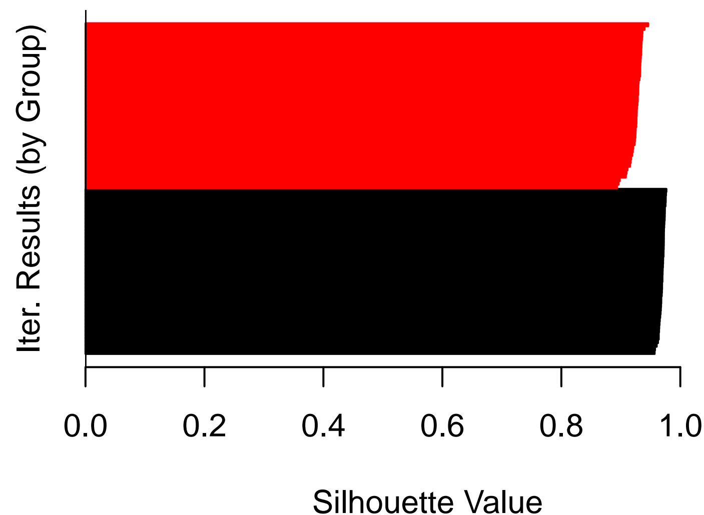
Once mutations were counted, it is possible to extract mutational signatures by NMF, using the `decipherMutationalProcesses()` function. This function takes two arguments:

1. a mutationCounts object (that we generated before)
2. a mutFrameworkParams object.

The latter stores all parameters for the NMF run, including the number of signatures to extract, and the total number of iterations. At each iteration, the original mutation counts are bootstrapped before factorization. Results from each iteration are aggregated. At the end of the analysis, a silhouette plot will show the

consistency of signatures extracted at each iteration. The default parameter values are a reasonably good starting point for most analyses. Here, we are only running 50 iterations (`num_totIterations = 50`), in order to make this first example run quicker. You should have ~500-1000 iterations for preparative NMF runs.

```
# how many signatures should we extract?
num.sign <- 2

# Define parameters for the non-negative matrix factorization procedure.
mouCancer.params <-
  setMutClusterParams(
    num.sign,                 # num signatures to extract
    num_totIterations = 50,  # bootstrapping: usually 500-1000
    num_parallelCores = 1)   # total num of cores to use (parallelization)

# Visualize parameters
mouCancer.params
##   mutFrameworkParams object - mutSignatures
##   num of params included: 16

# Extract new signatures
mouCancer.analysis <-
  decipherMutationalProcesses(input = allVCF.counts,
                              params = mouCancer.params)
```

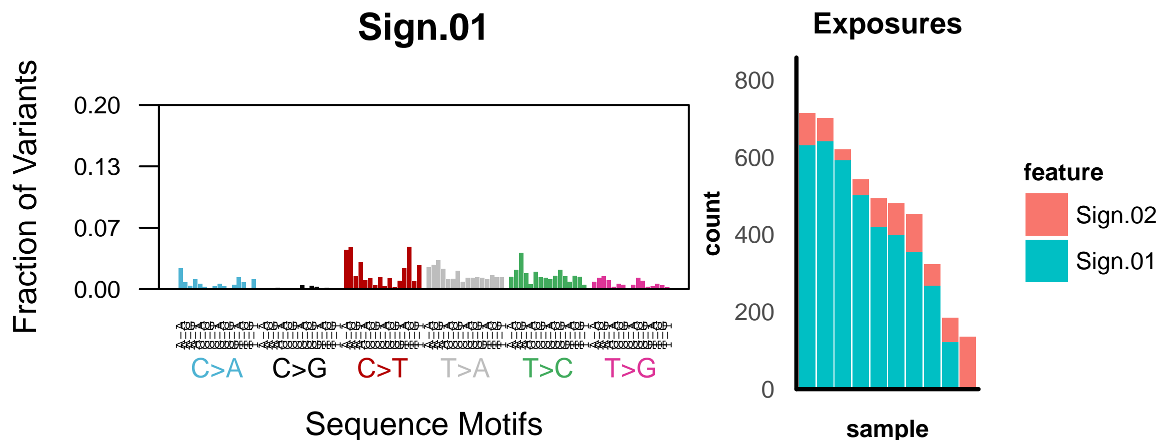



```
# Examine results
mouCancer.analysis$Results
## $signatures
##   Mutation Signatures object - mutSignatures
## ...
## $exposures
##   MutSignature Exposures object - mutSignatures
## ...
```

The results are returned as a list of lists. New signatures are included in the `Results$signatures` element. Exposures (how many mutations can be attributed to a given signature in each sample?) are stored in the `Results$exposures` element. The mutSignatures framework provides functions to visualize mutation profiles

of new signatures, as well as plot mutation counts by corresponding Signature from the exposure object.

```
# Retrieve and visualize the mutation profile of a signature
deNovo.signs <- mouCancer.analysis$Results$signatures

# Print to screen, or plot the object
print(deNovo.signs)
## Mutation Signatures object - mutSignatures
##
## Total num of Signatures: 2
## Total num of MutTypes: 96
## ...
plot(x = deNovo.signs, signature = 1)

# Visualize exposures
sample.expo <- mouCancer.analysis$Results$exposures

# Print to screen, or plot the object
print(sample.expo)
## MutSignature Exposures object - mutSignatures
##
## Total num of Samples: 10
## Total num of Signatures: 2  { first 2 signatures are displayed }
## Signature names:  Sign.01, Sign.02
## ...
plot(sample.expo) + ggplot2::ggtitle("Exposures")
```



Module 1: Data Import and Preparation

The following examples show how to import data from multiple sources (VCF files, MAF files, online repositories . . .) and assign three-nucleotide mutation types.

Importing mutation data via TCGAretriever

For more info about TCGAretriever (https://cran.r-project.org/web/packages/TCGAretriever/index.html), please check the TCGAretriever vignette (available soon!). Here, we are downloading the list of mutation found in the Ovarian Cancer TCGA provisional dataset, and use the *mutSignatures* package to compute DNA mutations in their three-nucleotide context for each patient.

```r
library(TCGAretriever)

# Get all genes with at least one mutation - pre-computed gene list - available online
ovcar.genes.url <- "http://www.labwizards.com/rlib/mutSignatures/ovcar.mutated.csv"
ovcar.genes <- read.csv(ovcar.genes.url, header = TRUE, as.is = TRUE)

# Use TCGAretriever to retrieve all mutations
tcgaRTV.mutations <- TCGAretriever::fetch_all_tcgadata(case_id = "ov_tcga_sequenced",
                                                       gprofile_id = "ov_tcga_mutations",
                                                       glist = ovcar.genes$SYMBOL,
                                                       mutations = TRUE)

# Nucleotide mutations data are included in the following attributes:
# "chr", "start_position", "end_position", and "variant_allele".
cols.of.interest <- c("gene_symbol", "case_id", "amino_acid_change",
                      "chr", "start_position", "reference_allele",
                      "variant_allele")
head(tcgaRTV.mutations[,cols.of.interest])
##     gene_symbol          case_id  aa_change chr  start_pos  ref_allele  var_allele
##  1          A2M  TCGA-04-1338-01     Q1051H  12    9230420           T           G
##  2         AAMP  TCGA-23-1117-01      K254*   2  219130790           T           A
##  3         AARS  TCGA-09-2049-01      D261H  16   70304134           C           G
##  4        ABCA1  TCGA-25-1635-01     E1253K   9  107576738           C           T
##  ...

# If these info are unavailable, you can extract mutation data
# from the "xvar_link" attribute, as follows:
tcgaRTV.mutations <- cbind(tcgaRTV.mutations,
                           extractXvarlinkData(tcgaRTV.mutations$xvar_link))

# The result is a data.frame including a series of new *Xvar columns
tcgaRTV.mutations[1:10, grepl("([sn]e_id|Xvar)$", colnames(tcgaRTV.mutations))]
##     entrez_gene_id          case_id  chrXvar    posXvar  refXvar  mutXvar
##  1               2  TCGA-04-1338-01       12    9230420        T        G
##  2              14  TCGA-23-1117-01        2  219130790        T        A
##  3              16  TCGA-09-2049-01       16   70304134        C        G
##  4              19  TCGA-25-1635-01        9  107576738        C        T
##  ...
```

Next, we filter out any mutation that is not a single-nucleotide variant; this means getting rid of insertions/deletions. You can study indels later, if interested. Missing bases (NA) won't be removed for now. Note that the `seq_colNames` argument provides the name of the columns carrying info about reference allele and mutated base.

Next, we use the `attachContext()` function to attach the tri-nucleotide context at the genomic location of each variant. BSgenome objects are accepted as reference genomes, such as 'hg19' (`BSgenome.Hsapiens.UCSC.hg19`) and 'mm10' (`BSgenome.Mmusculus.UCSC.mm10`). You need to specify:

- `start_colName`: name of the data.frame column providing info about the start.position
- `end_colName`: name of the data.frame column providing info about the end.position; for single nucleotide variants, the values in the `start_colName` and `end_colName` columns should match
- `nucl_contextN`: integer. Extent of the nucleotide context to attach, in bases
- `BSGenomeDb`, either "hg19" or "mm10". The corresponding package from bioconductor

(BSgenome.Hsapiens.UCSC.hg19 or BSgenome.Mmusculus.UCSC.mm10) is required. The nucleotide context is attached as the last column of the data.frame.

Before proceeding, let's make sure that the there is a perfect match between the experimental reference base and the base obtained from the reference assembly (mm10 and hg19). Next, let's use the information about SNV and nucleotide contexts to compute mutation types. The standard format is the one used by Alexandrov et al (for example A[T>G]T). When needed, reverse complement transformation will be automatically applied. Missing mutations (NAs) are removed during this phase. Mutations are attached as the last column in the data.frame.

The last step in data preparation is to count the different mutationTypes in each patient/sample. This can be achieved using the `countMutTypes()` function. This takes the following arguments:

- `mutTable`: input data.frame
- `mutType_colName`: name of the column that stores the mutation types
- `sample_colName`: name of the column that stores sample/patient identifiers; if omitted (NULL), all mutations are pooled as if they come from the same sample

Result are returned in a mutationCounts-class object. This can be used as is for the following steps, of it can be coerced to data.frame using `as.data.frame()` or `getCounts()`.

```
tcgaRTV.mutCounts <- countMutTypes(mutTable = tcgaRTV.mutations,
                                   mutType_colName = "mutType",
                                   sample_colName = "case_id")
tcgaRTV.df <- getCounts(tcgaRTV.mutCounts)

# Visualize the mutation count data.frame
tcgaRTV.df[1:6,1:6]
##           TCGA-**01  TCGA-**02  TCGA-**03  TCGA-**04  TCGA-**05  TCGA-**06
##   A[C>A]A         2          1          1          0          0          0
##   A[C>A]C         4          4          3          0          0          2
##   A[C>A]G         0          0          1          0          0          1
##   A[C>A]T         3          1          1          0          0          1
##   A[C>G]A         3          0          1          0          0          3
##   A[C>G]C         1          1          1          0          0          1
```

Importing mutation data from VCF files

Information about DNA variants are often stored in VCF files. VCF files can be loaded using the `read.delim()` function, and mutations can be accessed by looking at the following columns: CHROM, POS, REF, ALT. Similarly to the previous example, VCF files can be imported and processed by mutSignatures. Finally, mutation types are counted. In this case, there is only one sample (in this example, only one VCF file is processed). Therefore, the `sample_colName` is not specified.

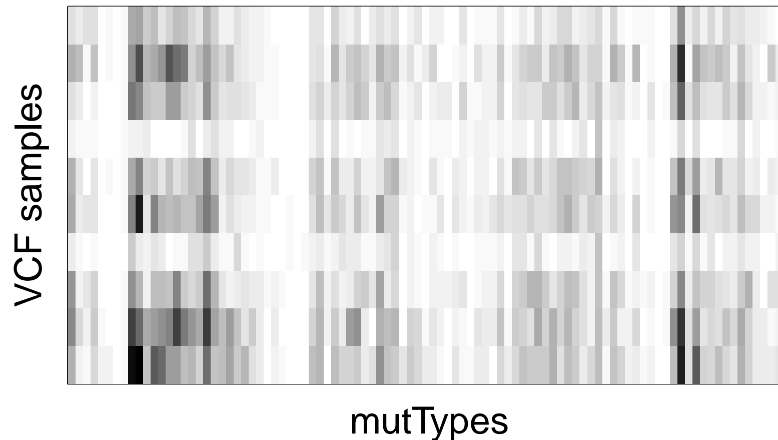
```
# Download some VCF data for testing
vcf_url <- "http://www.labwizards.com/rlib/test_muts.tar.gz"
if (!"test_muts.tar.gz" %in% dir()){
  download.file(vcf_url, destfile="test_muts.tar.gz")
}
untar(tarfile = "test_muts.tar.gz", compressed = "gzip", list = F)
vcfFiles <- grep("^0[01][0-9]\\.vcf$", dir(), value = TRUE)

# Import the data in R
vcf.head <- c("CHROM", "POS", "ID", "REF", "ALT", "QUAL",
              "FILTER", "INFO", "FORMAT", "SAMPLEID")
```

```
vcfData_01 <- read.delim(vcfFiles[1], comment.char = '#',
                      header = F, col.names = vcf.head)

# Filter SNVs, attach context, rimove Mismatches, and attach MutTypes
vcfData_01 <- filterSNV(dataSet = vcfData_01,
                        seq_colNames = c("REF", "ALT"))
mm10 <- BSgenome.Mmusculus.UCSC.mm10::BSgenome.Mmusculus.UCSC.mm10
vcfData_01 <- attachContext(mutData = vcfData_01,
                            chr_colName = "CHROM",
                            start_colName = "POS",
                            end_colName = "POS",
                            nucl_contextN = 3,
                            BSGenomeDb = mm10)
vcfData_01 <- removeMismatchMut(mutData = vcfData_01,
                                refMut_colName = "REF",
                                context_colName = "context",
                                refMut_format = "N")
vcfData_01 <- attachMutType(mutData = vcfData_01,
                            ref_colName = "REF",
                            var_colName = "ALT",
                            context_colName = "context")
vcf.tab_01 <- countMutTypes(mutTable = vcfData_01, mutType_colName = "mutType")
```

For repeating all these operations for each VCF file, it is recommended to use the `importVCFfiles()`, and `processVCFdata()` functions. These perform the operations described above on a series of VCF files. The first function iterates through a series of VCF files and imports the corresponding mutations. The `SAMPLEID` column is attached as the last column of the data.frame to keep track of the VCF file where a variant was found. The second function reads each variant and computes the corresponding three-nucleotide mutation type. The default behavior is to include all mutations in a single data.frame. When a very large number of variants are processed together, it may be faster to specify the `sample_colName="SAMPLEID"` argument. This forces mutations to be processed in multiple iterations (sample- or patient-wise approach). The final results are aggregated and returned as a single data.frame.

```
# Load and process multiple files
allvcfData <- importVCFfiles(vcfFiles = vcfFiles)
mm10 <- BSgenome.Mmusculus.UCSC.mm10::BSgenome.Mmusculus.UCSC.mm10
fullData <- processVCFdata(vcfData = allvcfData,
                           sample_colName = "SAMPLEID",
                           BSGenomeDb = mm10)

# Count mutation types
allVCF.counts <- countMutTypes(mutTable = fullData,
                               sample_colName = "SAMPLEID")

# Coerce to data.frame
cntDF <- getCounts(allVCF.counts)

# Coerce to numeric matrix
cntMAT <- as.matrix(allVCF.counts)

# Ultra-simple visualization as image
graphics::image(cntMAT,
                col = grDevices::colorRampPalette(colors = c("white", "black"))(100),
                xlab = "", ylab = "", las = 2, main = "Mut Type Heatmap - mut. Counts",
```

```
                 xaxt = 'n', yaxt = 'n', mar = c(1, 1, 1, 1))
graphics::title(ylab="VCF samples", line=0.5, cex.lab=1.2)
graphics::title(xlab="mutTypes", line=0.5, cex.lab=1.2)
```



Importing mutation data from MAF files

The Broad Institute and other Cancer Research Centers release somatic mutation collections in MAF or similar rich non-VCF format. A single MAF file usually includes information about genetic variants, as well as many clinical parameters of a large number of patients in a single text file. Importing an processing mutation counts from MAF or similar files by `mutSignatures` proceeds according to the pipeline shown below. The only reccomandation is to explore the input file before startingm in order to make sure that the correct information is selected at each step.

```
# define URL, Broad Nozzle Report - Ovarian Cancer Mutation Report - MAF
MAF_url <- c("http://gdac.broadinstitute.org/runs/",
             "analyses__2016_01_28/reports/cancer/OV/",
             "MutSigNozzleReport2CV/OV-TP.final_analysis_set.maf")
MAF_url <- paste(MAF_url, collapse = "")

download.file(MAF_url, destfile="input_mut.maf")
maf.dset <- read.delim("input_mut.maf", header = TRUE, as.is = TRUE)

# Adjust TCGA identifiers
maf.dset$TCGAid <- substr(maf.dset$Tumor_Sample_Barcode, 1, 15)

# Filter single nucleotide variants
maf.dset <- filterSNV(dataSet = maf.dset,
                      seq_colNames = c("Reference_Allele",
                                       "Tumor_Seq_Allele1",
                                       "Tumor_Seq_Allele2"))

# Attach 3-nucleotide context
hg19 <- BSgenome.Hsapiens.UCSC.hg19::BSgenome.Hsapiens.UCSC.hg19
maf.dset <- attachContext(mutData = maf.dset,
                          chr_colName =  "Chromosome",
                          start_colName = "Start_Position",
```

```
                         end_colName = "End_position",
                         nucl_contextN = 3,
                         BSGenomeDb = hg19)

# Remove mismatched positions
maf.dset <- removeMismatchMut(mutData = maf.dset,
                             refMut_colName = "Reference_Allele",
                             context_colName = "context",
                             refMut_format = "N")

# Attach mutation Type
maf.dset <- attachMutType(mutData = maf.dset,
                        ref_colName = "Reference_Allele",
                        var_colName = "Tumor_Seq_Allele1",
                        var2_colName = "Tumor_Seq_Allele2",
                        context_colName = "context")

# Attach mutation Type
maf.counts <- countMutTypes(mutTable = maf.dset,
                           sample_colName = "TCGAid",
                           mutType_colName = "mutType")

# Mutation Counts object - mutSignatures
maf.counts
##  Mutation Counts object - mutSignatures
##
##  Total num of MutTypes: 96
##  MutTypes: A[C>A]A, A[C>A]C, A[C>A]G, A[C>A]T, A[C>G]A ...
##
##  Total num of Samples: 466
##  Sample Names: TCGA-23-1029-01, TCGA-42-2589-01, TCGA-59-2352-01, ...
```

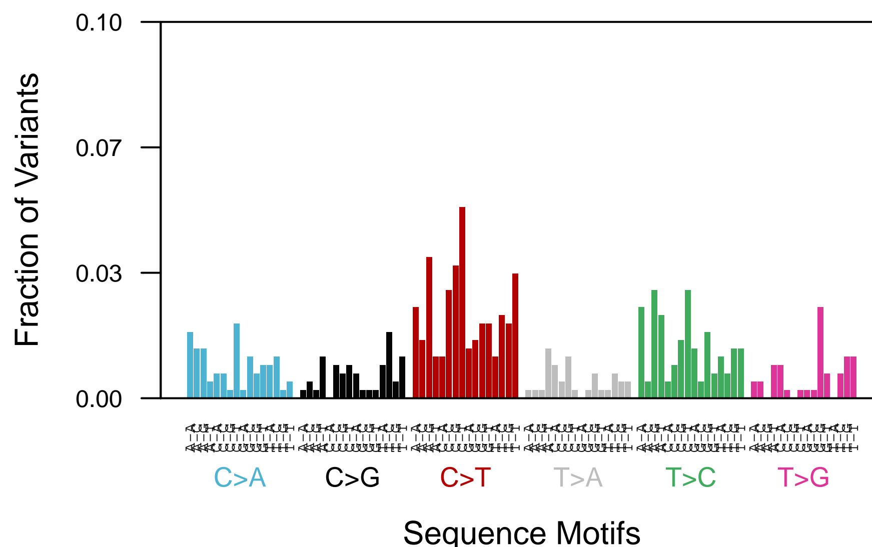
Importing mutation data from Sequenza

Sequenza is an R-based framework that can be used for imputing allele-specific copy number and mutation profiles from tumor sequencing data doi:10.1093/annonc/mdu479. The *mutSignatures* framework can import mutation data that were processed via the Sequenza pipeline. In the Sequenza output, SNV are specified according to the following format: `A>G`. The steps to process this type of data are very similar to what discussed above. The key difference is the `refMut_format` argument passed to the `removeMismatchMut()` function. In this case, `refMut_format = "N>N"`. An example is shown below.

```
library(sequenza)

data.file <- system.file("data", "example.seqz.txt.gz", package = "sequenza")
seqz.df <- sequenza.extract(data.file)

# Explore Sequenza Data
head(seqz.df$mutations[[1]])
##     chromosome  position  good.reads  adjusted.ratio      F  mutation
##  364          1  10436585         158       0.9956093  0.215       C>T
##  386          1  11140488          41       0.9956093  0.122       A>G
##  510          1  12888791          78       0.9956093  0.154       A>G
```

```r
seqz.muts <- data.frame(do.call(rbind,
                                lapply(seqz.df$mutations, function(x) {
                                  x[grepl("^[ACTG]>[ACTG]$", x$mutation),]})),
                        stringsAsFactors = FALSE, row.names = NULL)

# Attach 3-nucleotide context
hg19 <- BSgenome.Hsapiens.UCSC.hg19::BSgenome.Hsapiens.UCSC.hg19
maf.dset <- attachContext(mutData = seqz.muts,
                          chr_colName =  "chromosome",
                          start_colName = "position",
                          end_colName = "position",
                          nucl_contextN = 3,
                          BSGenomeDb = hg19)

# Remove mismatched positions <- Note the refMut_format argument HERE!
maf.dset <- removeMismatchMut(mutData = maf.dset,
                              refMut_colName = "mutation",
                              context_colName = "context",
                              refMut_format = "N>N")

# Attach mutation Type
maf.dset <- attachMutType(mutData = maf.dset,
                          ref_colName = "mutSite.dnaSeq.Ref",
                          var_colName = "mutSite.dnaSeq.Mut",
                          context_colName = "context")

# Attach mutation Type
maf.counts <- countMutTypes(mutTable = maf.dset,
                            mutType_colName = "mutType")
```

```r
plot(maf.counts, sample = 1, main = "Sequenza Input", ylim = c(0, 0.1))
```



Importing mutation data from pre-computed mutation type matrices

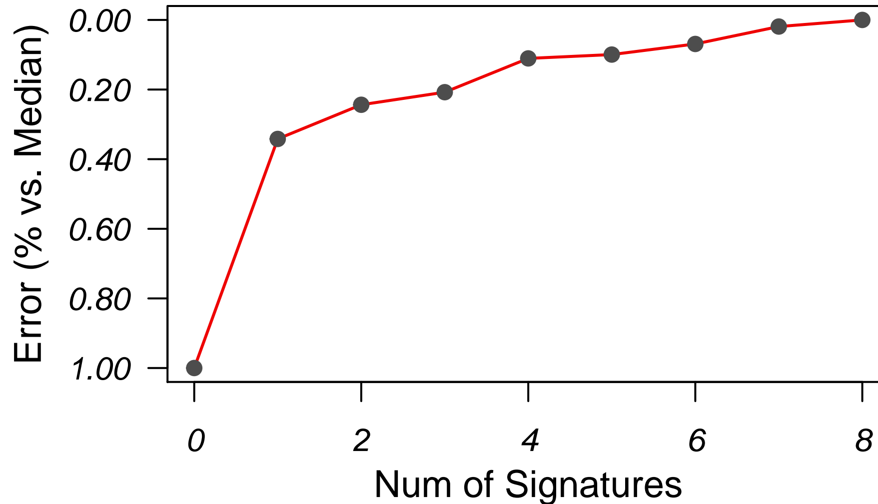
If a mutation count matrix is available and ready to use, you can easily coerce it to a mutation counts object, using `as.mutation.counts()`. The following example shows how to download a mutation count matrix from the *sanger.ac.uk* repository and coerce it to a *mutationCounts* object.

```
ovcar.counts.url <- paste("ftp://ftp.sanger.ac.uk/pub/cancer/AlexandrovEtAl/",
                          "mutational_catalogs/exomes/Ovary/",
                          "Ovary_exomes_mutational_catalog_96_subs.txt", sep = "")
ovcar.counts.data <- read.delim(ovcar.counts.url,
                                as.is = TRUE,
                                header = TRUE,
                                row.names = "Mutation.Type")
# Display data.frame
ovcar.counts.data[1:20,1:9]
##          ID1  ID2  ID3  ID4  ID5  ID6  ID7  ID8  ID9
##  A[C>A]A    0    0    0    0    0    1    0    0    0
##  A[C>A]C    0    0    0    0    0    0    0    0    0
##  A[C>A]G    0    0    0    0    0    0    1    0    2
##  A[C>A]T    0    0    0    0    0    0    0    0    1
##  ...

# Keep only genomes with at least 30 mutations
KEEP <- as.logical(apply(ovcar.counts.data, 2, sum) >= 30)
ovcar.counts.data <- ovcar.counts.data[, KEEP]

# Import via as.mutation.counts
ocd <- as.mutation.counts(ovcar.counts.data)
ocd
##  Mutation Counts object - mutSignatures
##
##   Total num of MutTypes: 96
##   MutTypes: A[C>A]A, A[C>A]C, A[C>A]G, A[C>A]T, A[C>G]A ...
##
##   Total num of Samples: 471
##   Sample Names: OCC01PT, OCC02PT, OCC03PT, OCC04PT, ...
```

Module 2. Mutation Signature Extraction

The step following data preparation is *de novo* signature extraction by NMF. One of the key parameter for this step is the number of signatures to extract. As a rule of thumb, the number of signatures should be much lower than the total number of cases. It is very difficult to estimate *a priori* how many processes (ie, signatures) are active in a group of tumors or samples. However, mutSignatures comes with a function that addresses this issue. Briefly, the `prelimProcessAssess()` function extracts an increasing number of signatures using no bootstrapping and fast sub-optimal parameters. While the results are usually inaccurate, the function tracks changes in total error with respect to the number of extracted signatures. This can help deciding about a reasonable number of signatures to extract. The `approach` parameter can be set to `"freq"` to force sample-wise mutation counts normalization. For large datasets, this step may take several minutes to complete.

```
mouCancer.assess <- prelimProcessAssess(input = ocd,
                                        approach = "counts")
```



The function returns a data.frame and a plot showing changes in total error with respect to the number of extracted signatures *!this feature is still under development, at the moment*. In this case, the first extracted signature explains most of the error (compared to median mutation counts). Extracting more signatures further minimizes the total error. Here, we will *de novo* extract 3 signatures. To extract signatures, we first have to set a list of parameters that will control the non-negative matrix factorization. These include:

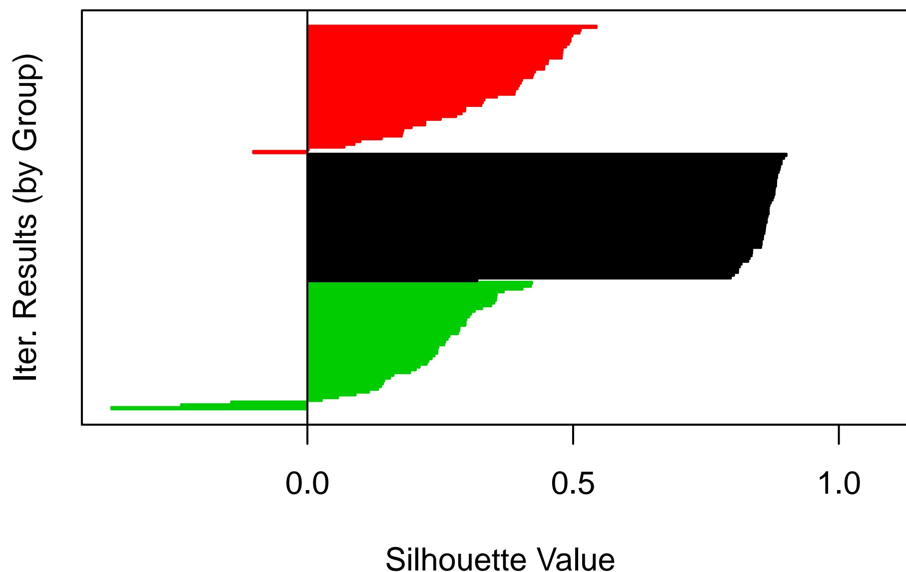
- `num_processesToExtract`: number of signatures to extract
- `approach`: "counts" vs. "freq". This is used to use counts (standard) or force sample-wise normalization. The standard approach will give more weight to samples carrying more mutations.
- `num_totIterations`: total number of bootstrapping iterations. Typically 500-1000. For preliminary analysis, a number of 50-100 is usually sufficient.
- `num_parallelCores`: number of cores. Setting a value bigger than 1 means using parallelization.
- `algorithm`: "alexa" for using the standard Brunet NMF algorithm vs "lin" for using the NMF approach proposed by Chih-Jen Lin doi:10.1109/TNN.2007.895831
- `debug`: TRUE for visualizing convergence plots and messages during NMF execution.

The NMF extraction is then executed by `decipherMutationalProcesses()`. The function takes two arguments: the parameters (mutFrameworkParams-class) and the mutation counts object (mutationCounts-class). A silhouette plot is returned at the end of this analysis. The plot provides a summary of whether the boostrapping operation resulted in consistent results. Briefly, each group of bars (identified by different colors) is linked to a newly extracted signature. Silhouettes with high profile (values close to 1) are suggestive of consistent results across several bootstrapping iterations (strong, reproducible signature). Low silhouette profiles (values close to 0 or negative values) suggest unreliable results (unreliable signature). This could be due to the high variability of mutational profiles across the samples being analyzed, or to a poor choice of the number of signatures to extract (try increasing or decreasing *num_processesToExtract*).

The NMF analysis results are returned as a list of lists: *de novo* extracted signatures are included in the `Results$signatures` element; imputed exposures are included in the `Results$exposures` element. Both objects can be coerced to data.frame or used *as is* for visualization or further analyses.

```
# And define parameters for the non-negative matrix factorization procedure
ocd.params <- setMutClusterParams(num_processesToExtract = 3,
                                  approach = "counts",
                                  num_totIterations = 50,
```

```
                                            num_parallelCores = 4,    # set to 1 to avoid parallelization
                                            debug = FALSE,
                                            algorithm = "alexa")
# Now GO! Will take a while
ocd.analysis <- decipherMutationalProcesses(input = ocd,
                                            params = ocd.params)
```

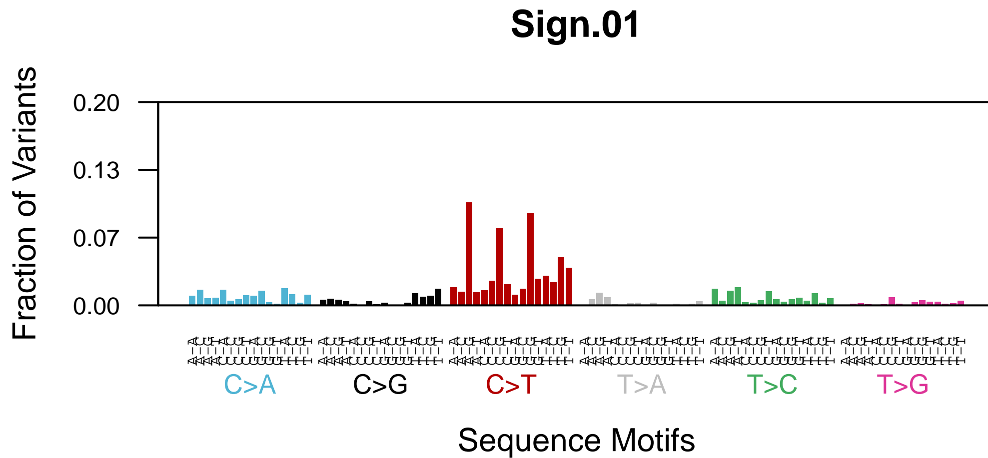



```
# New signatures
ocd.signs <- ocd.analysis$Results$signatures
as.data.frame(ocd.signs)[1:10,]
##              Sign.01     Sign.02     Sign.03
##  A[C>A]A 0.007574636 0.022128604 0.016528003
##  A[C>A]C 0.007718059 0.021140710 0.026975176
##  A[C>A]G 0.003880905 0.003242074 0.002508504
##  A[C>A]T 0.005436846 0.016271546 0.011834253
##  A[C>G]A 0.001260808 0.028931052 0.009241104
##  ...

# Corresponding exposures
ocd.expos <- ocd.analysis$Results$exposures
as.data.frame(ocd.expos)[1:10,]
##                   Sign.01      Sign.02  Sign.03
##  OCC06PT        94.185922 2.261763e-16 14.81408
##  TCGA.04.1331.. 19.701404 2.298506e+01 13.31354
##  TCGA.04.1336.. 23.225619 1.756050e+01 16.21388
##  TCGA.04.1338.. 47.108898 2.243059e+01 47.46051
##  TCGA.04.1343.. 21.690725 1.389099e+01 19.41829
##  ...
```

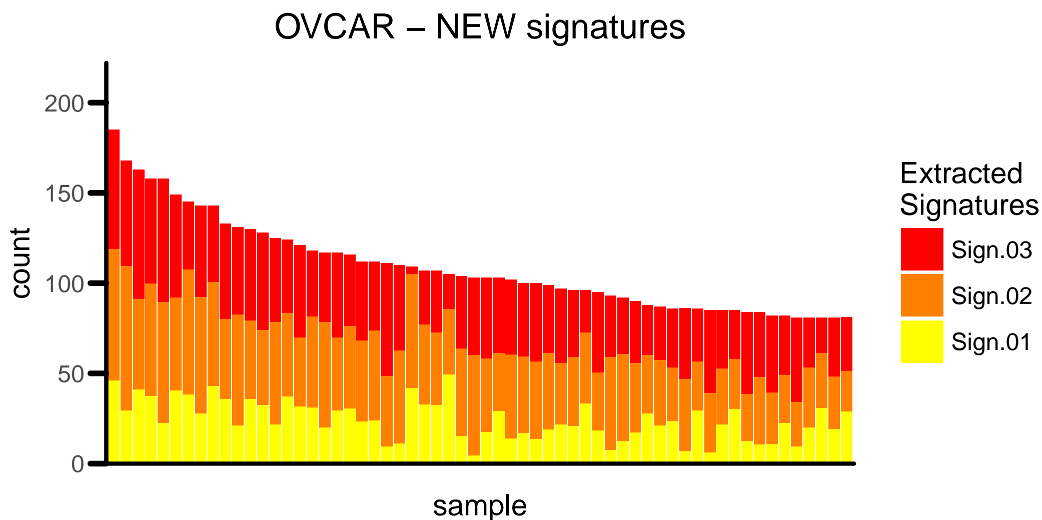
Module 3. Visualization of Signatures and Exposures

Mutation Profiles and Signature Exposures

The *mutSignatures* framework comes with a series of tools for visualizing the results of a signature extraction analysis. Mutational signatures are vectors of frequencies of each possible DNA mutation that includes information about the surrounding three-nucleotide context (total n=96). Likewise, mutation counts are vectors of discrete counts of the same mutations, and can be easily converted to frequencies. Both mutational signatures and mutation counts can be visualized in a similar fashion, and using the `plot()` method, which in turn calls the `plotMutTypeProfile()` function. Examples are provided. It is sufficient to specify the numeric index of the signature or the index we want to plot. Alternatively, the identifier of the sample can be set as the `sample` argument.



Exposure objects store information about how individual signatures contribute to the total number of SNVs found in a series of biological samples or patients. In other words, exposure objects provide information about how many SNVs found in a given sample can be attributed to biologic processes associated to the different signatures. According to matrix algebra notation, the NMF factorizes a mutation count matrix, namely **V**, in two matrices so that **V = W x H**, where W is the mutation signature matrix, and H corresponds to the exposure matrix (here, transposed). Usually, exposure matrices are effectively visualized as stacked barplots. Here, the plot obtained calling the `plot()` methos uses functions from the `ggplot2` library.

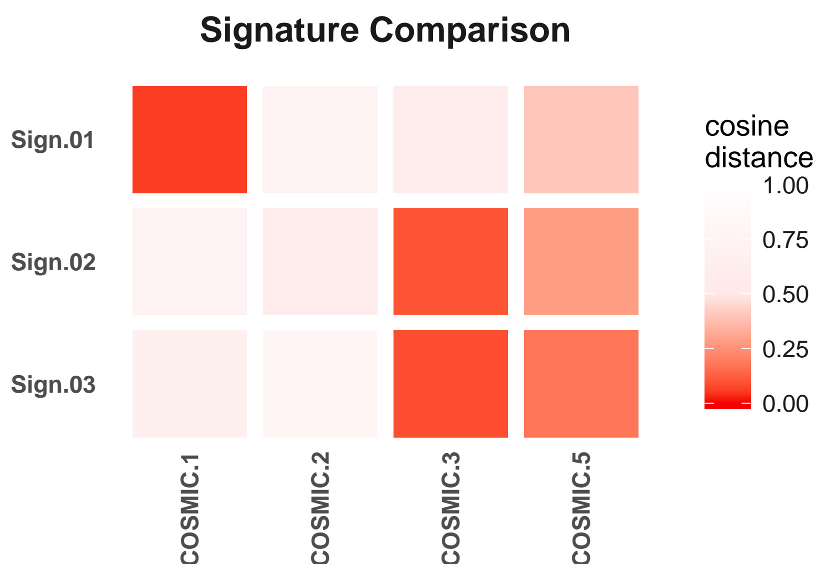


Matching Mutational Signatures

A useful tool provided with the mutSignatures package is the `matchSignatures()` function, which allows to compare mutational signatures from two *mutationSignatures* objects. If only one *mutational Signatures* object is passed as argument, by default the mutation signatures are compared to all COSMIC signatures. The comparison is performed by computing the *cosine* distance between each pair of signatures. An heatmap highlighting the closest mutSignatures matches is also returned. An example is provided.

```
# Retrieve COSMIC signatures from online repo, and then subset
cosmix <- getCosmicSignatures()
cosmix.ovcar <- cosmix[c(1, 2, 3, 5)]

# match OVcar and COSMIC signatures
mSign.ov <- matchSignatures(mutSign = ocd.signs, reference = cosmix.ovcar)
```

```
print(mSign.ov$plot)
```



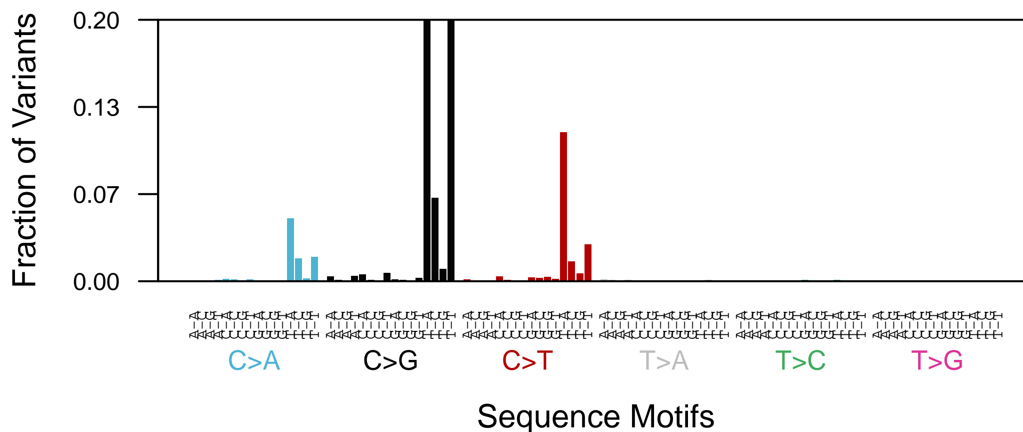
Deconvoluting mutation counts agains known signatures

An important type of mutational signature analysis does not require the extraction of novel signatures. A simple yet very powerful analysis is the deconvolution of somatic mutation catalogs against a deck of already-established mutational signatures (such as the COSMIC signatures, initially described by Alexandrov LB et al). Mutation signature exposures can be computed by solving the matrix algebra equation **V** ~ **W x H**, where V and W are known and H is unknown. The *mutSignatures* framework supports this kind of analysis. Functions from Module 1 can be used to import and prepare the mutation counts data, and then the `resolveMutSignatures()` function is employed to deconvolute mutation counts. This function relies on the `fcnnls()` function from the `NMF` package. Please, note that the `fcnnls()` function requires the `corpcor` package to be installed; therefore, make sure to install this package before running `resolveMutSignatures()`. The key arguments passed to this function are:

- `mutCountData`: mutationCounts object
- `signFreqData`: mutationSignatures object

The *mutSignatures* framework also comes with a function to retireve the most current list of COSMIC signatures and coerce it to a mutationSignatures object. This is achieved via the `getCosmicSignatures()` function.

```r
# Make sure that `corpcor` is installed
if (!"corpcor" %in% rownames(installed.packages()))
  install.packages("corpcor")

# Obtain COSMIC signatures
cosmix <- getCosmicSignatures()
print(cosmix)
##  Mutation Signatures object - mutSignatures
##
##  Total num of Signatures: 30
##  Total num of MutTypes: 96
##  ...

# Coerce to data.frame
as.data.frame(cosmix)
##           Sign.1  Sign.2  Sign.3  Sign.4  Sign.5  Sign.6
##  A[C>A]A 1.1e-02 6.8e-04 0.02217 0.03650 0.01494 0.00170
##  A[C>A]C 9.1e-03 6.1e-04 0.01787 0.03090 0.00896 0.00280
##  A[C>A]G 1.4e-03 9.9e-05 0.00213 0.01830 0.00220 0.00050
##  A[C>A]T 6.2e-03 3.2e-04 0.01626 0.02430 0.00920 0.00190
##  ...

# Subset
cosmix[c(1, 3, 5, 13)]
##  Mutation Signatures object - mutSignatures
##
##   Total num of Signatures: 4
##   Total num of MutTypes: 96
##   ...
```

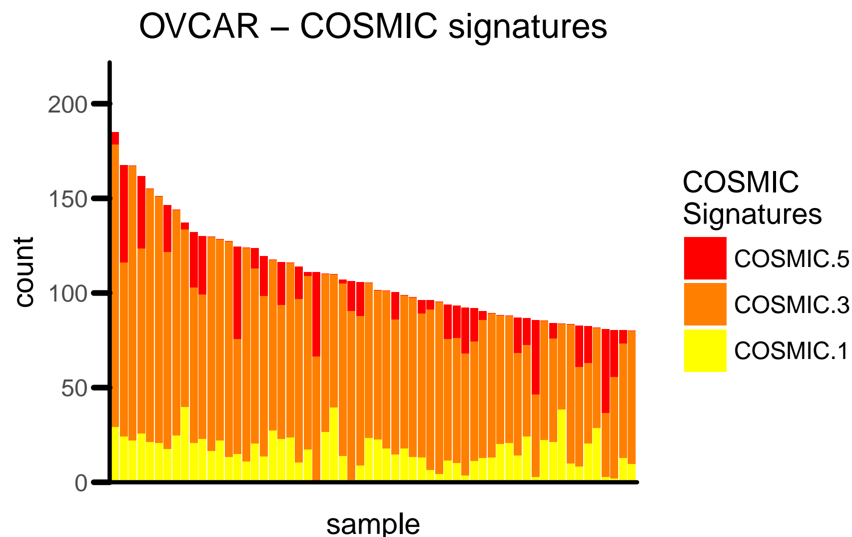


The results are returned as a list of lists. The exposure object is included in the `results` element of the output list. Specifically, the `results$count.result` element includes deconvoluted mutation counts. Conversely, the `results$freq.result` includes the same results converted to frequency (signature contributions sum up to unity in each sample). By default, the `plot()` method applied to the exposure object will return a stacked barplot, as seen before.

```r
# MutSignature Exposures object - mutSignatures (counts)
deconvoluted.signs$results$count.result
```

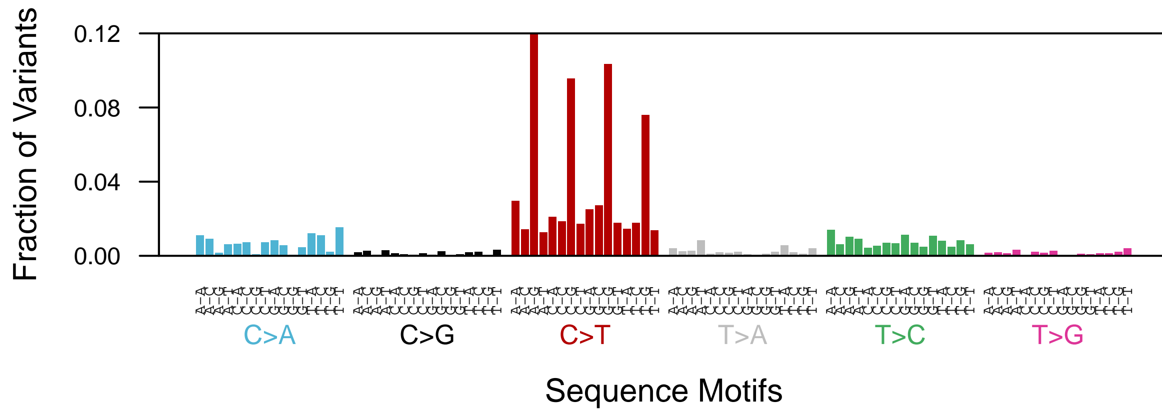
```
##  MutSignature Exposures object - mutSignatures
##  ...
##    Sign.1   Sign.2   Sign.3
##    ------   ------   ------
##  +      9        0        6  +  OCC01PT
##  +      9        3        1  +  OCC02PT
##  +     18        0        0  +  OCC03PT
##  +      7        0        0  +  OCC04PT
##  ...

# MutSignature Exposures object - mutSignatures (freqs)
deconvoluted.signs$results$freq.result
##  MutSignature Exposures object - mutSignatures
##  ...
##    Sign.1   Sign.2   Sign.3
##    ------   ------   ------
##  +  0.59     0.00     0.41   +  OCC01PT
##  +  0.71     0.25     0.04   +  OCC02PT
##  +  1.00     0.00     0.00   +  OCC03PT
##  +  1.00     0.00     0.00   +  OCC04PT
##  ...

# print signatures identifiers
getSignatureIdentifiers(sample.counts)
#[1] "Signature.1" "Signature.3" "Signature.5"
```

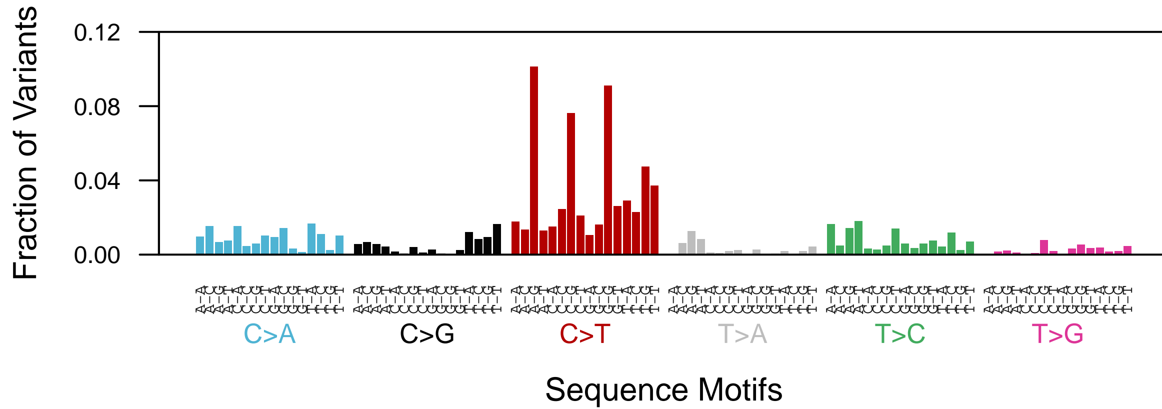
```
# Plot exposures
new.p <- plot(x = deconvoluted.signs$results$count.result, top = 60)
my.feats <- unique(new.p$data$feature)
new.p +
  ggplot2::ggtitle(label = "OVCAR - COSMIC signatures") +
  ggplot2::scale_fill_manual(name = "COSMIC\nSignatures",
                             values = grDevices::heat.colors(length(my.feats)))
```

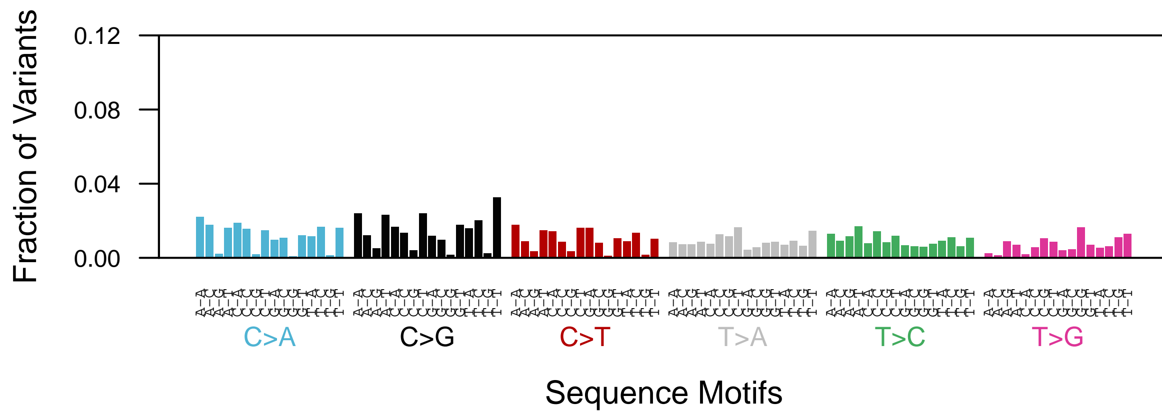


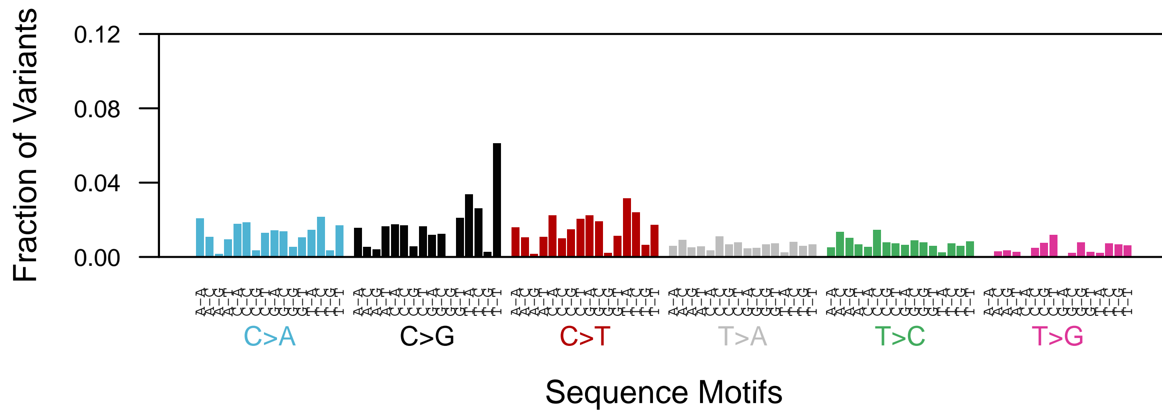
In the previous example, the following COSMIC signatures were used for the deconvolution: COSMIC *#1*, *#3*, and *#5*. We used these signatures, since they were identified as those operative in Ovarian Cancers by the work of the Sanger Institute (http://cancer.sanger.ac.uk/cosmic/signatures) and the TCGA consortium

(https://cancergenome.nih.gov/). As proof of principle, our demo analysis (see above) conducted on the `ocd` object (Mutation Count object summarizing mutations found in 323 ovarian cancer specimens, from the supplementary materials of the Alexandrov LB et al) resulted in the identification of novel mutational signatures that are very similar to COSMIC *#1*, *#3*, and *#5*.









Session Info

```
## [1] "Elapsed time for building the vignette and running all examples: 36.4 mins"
```

```r
print(sessionInfo())
```

```
## R version 3.4.3 (2017-11-30)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.3 LTS
##
## Matrix products: default
## BLAS: /usr/lib/libblas/libblas.so.3.6.0
## LAPACK: /usr/lib/lapack/liblapack.so.3.6.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
```

```
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats     graphics  grDevices utils     datasets  methods
## [8] base
##
## other attached packages:
## [1] corpcor_1.6.9       sequenza_2.1.2      squash_1.0.8
## [4] copynumber_1.18.0   TCGAretriever_1.3   mutSignatures_1.3.7
## [7] Biobase_2.38.0      BiocGenerics_0.24.0 ggplot2_2.2.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_0.12.15                      lattice_0.20-35
##  [3] Rsamtools_1.30.0                  Biostrings_2.46.0
##  [5] rprojroot_1.3-2                   digest_0.6.15
##  [7] BSgenome.Mmusculus.UCSC.mm10_1.4.0 foreach_1.4.4
##  [9] gridBase_0.4-7                    R6_2.2.2
## [11] GenomeInfoDb_1.14.0               plyr_1.8.4
## [13] backports_1.1.2                   stats4_3.4.3
## [15] evaluate_0.10.1                   pracma_2.1.4
## [17] httr_1.3.1                        pillar_1.1.0
## [19] zlibbioc_1.24.0                   rlang_0.1.6
## [21] lazyeval_0.2.1                    curl_3.1
## [23] S4Vectors_0.16.0                  Matrix_1.2-11
## [25] rmarkdown_1.8                     labeling_0.3
## [27] BiocParallel_1.12.0               stringr_1.3.0
## [29] RCurl_1.95-4.10                   munsell_0.4.3
## [31] proxy_0.4-21                      DelayedArray_0.4.1
## [33] compiler_3.4.3                    rtracklayer_1.38.3
## [35] pkgmaker_0.22                     htmltools_0.3.6
## [37] SummarizedExperiment_1.8.1        tibble_1.4.2
## [39] GenomeInfoDbData_1.0.0            IRanges_2.12.0
## [41] codetools_0.2-15                  matrixStats_0.53.0
## [43] XML_3.98-1.9                      BSgenome.Hsapiens.UCSC.hg19_1.4.0
## [45] GenomicAlignments_1.14.1          bitops_1.0-6
## [47] grid_3.4.3                        xtable_1.8-2
## [49] gtable_0.2.0                      registry_0.5
## [51] magrittr_1.5                      scales_0.5.0
## [53] stringi_1.1.7                     XVector_0.18.0
## [55] reshape2_1.4.3                    doParallel_1.0.11
## [57] RColorBrewer_1.1-2                NMF_0.21.0
## [59] iterators_1.0.9                   tools_3.4.3
## [61] BSgenome_1.46.0                   rngtools_1.2.4
## [63] yaml_2.1.16                       colorspace_1.3-2
## [65] cluster_2.0.6                     GenomicRanges_1.30.1
## [67] knitr_1.19
```