Overview.

Point mutations occurring in a genome can be divided into 96 categories based on the base being mutated, the base it is mutated into and its two flanking bases. Therefore, for any patient, it is possible to represent all the point mutations occurring in that patient’s tumor as a vector of length 96, where each element represents the count of mutations for a given category in the patient.

A mutational signature represents the pattern of mutations produced by a mutagen or mutagenic process inside the cell. Each signature can also be represented by a vector of length 96, where each element represents the probability that this particular mutagenic process generates a mutation of the 96 above mentioned categories. In this R package, we provide a set of functions to extract and visualize the mutational signatures that best explain the mutation counts of a large number of patients.

In this vignette, we give an overview of the package by presenting some of its main functions.
1 Changelog

1.0.0 package released on Bioconductor in May 2018.

2 Algorithms and useful links

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Extended name</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SparseSignatures</td>
<td>De Novo Mutational Signature Discovery in Tumor Genomes using SparseSignatures</td>
<td>Publication</td>
</tr>
</tbody>
</table>

3 Using the SparseSignatures R package

We now present the main features of the package. To start, we show how to load data and transform them to a count matrix to perform the signatures discovery; first we load some example data provided in the package.

```r
library("SparseSignatures")
## Loading required package: NMF
## Loading required package: pkgmaker
## Loading required package: registry
## Attaching package: 'pkgmaker'
## The following object is masked from 'package:base':
## isFALSE
## Loading required package: rngtools
## Loading required package: cluster
```
Extracting sparse mutational signatures via LASSO

```r
## NMF - BioConductor layer [OK] | Shared memory capabilities [NO: synchronicity] | Cores 19/20
## To enable shared memory capabilities, try: install.extras('NMF')

data(ssm560_reduced)
head(ssm560_reduced)
## sample chrom pos ref alt
## 1: PD10014a 1 186484577 A C
## 2: PD10014a 7 141761948 G A
## 3: PD10014a 7 71266228 C T
## 4: PD10014a 8 82304475 A T
## 5: PD10014a 3 191275626 T A
## 6: PD10014a 4 135265376 C T
```

These data are a reduced version with only 3 patients of the 560 breast tumors provided by Nik-Zainal, Serena, et al. (2016). We can transform such input data to a count matrix to perform the signatures discovery with the function import.counts.data. To do so, we also need to specify the reference genome as a BSgenome object and the format of the 96 nucleotides to be considered. This can be done as follows, where in the example we use hs37d5 as our reference genome.

```r
library("BSgenome.Hsapiens.1000genomes.hs37d5")
## Loading required package: BSgenome
## Loading required package: S4Vectors
## Loading required package: stats4
## Attaching package: 'S4Vectors'
## The following object is masked from 'package:NMF':
## nr
## The following object is masked from 'package:pkgmaker':
## new2
## The following object is masked from 'package:base':
## expand.grid
## Loading required package: IRanges
## Loading required package: GenomeInfoDb
## Loading required package: GenomicRanges
## Loading required package: Biostrings
## Loading required package: XVector
```
Extracting sparse mutational signatures via LASSO

## Attaching package: 'Biostrings'

## The following object is masked from 'package:base':

## Loading required package: rtracklayer

bsg = BSgenome.Hsapiens.1000genomes.hs37d5
data(mutation_categories)
head(mutation_categories)

## context alt cat
## 1: A:A C>A A[C>A]A
## 2: C:A C>A C[C>A]A
## 3: G:A C>A G[C>A]A
## 4: T:A C>A T[C>A]A
## 6: C:A C>G C[C>G]A

imported_data = import.counts.data(input=ssm560_reduced,bsg=bsg,mutation_categories=mutation_categories)

## Warning in import.counts.data(input = ssm560_reduced, bsg = bsg, mutation_categories = mutation_categories): Some samples have fewer than 100 mutations:
## PD10010a, PD10011a, PD10014a

head(imported_data)

## PD10010a 37 25 8 24 35 5 16 25 49
## PD10011a 103 59 16 73 113 54 31 102 116
## PD10014a 235 241 37 234 158 71 26 180 229
## PD10010a 31 100 42 21 15 17 30 48 20
## PD10011a 73 228 109 61 70 56 165 184 116
## PD10014a 89 178 186 105 90 126 174 261 122
## PD10010a 29 44 8 6 10 23 34 28 8
## PD10011a 113 169 77 41 73 105 105 75 30
## PD10014a 167 211 76 27 84 59 244 238 35
## PD10010a 23 15 19 20 26 48 37 55 43
## PD10011a 102 60 37 22 65 71 52 108 103
## PD10014a 243 107 105 40 144 136 124 144 197
## PD10010a 12 7 18 16 14 17 20 30 6
## PD10011a 116 80 89 103 103 78 102 158 40
## PD10014a 116 139 145 145 217 103 144 112 129 47
## PD10010a 8 5 13 31 22 11 22 6 12
## PD10011a 65 55 188 78 50 14 55 55 66
## PD10014a 54 70 107 146 126 24 160 63 70
## PD10010a 9 14 40 32 82 25 6 6 6

4
Extracting sparse mutational signatures via LASSO

The function import.counts.data can also take a text file as input with the same format as the one shown above. Now, we show an example of a visualization feature provided by the package, and we show the counts for the first patient PD10010a in the following plot.

After the data are loaded, signatures can be discovered. To do so, we need to define a set of parameters on which to perform the estimation.

First of all, we need to specify the ranges for the number of signatures (variable K) and the LASSO penalty value (variable lambda rate) to be considered. The latter is more complicated to estimate, as it requires that the values in the range not to be too small in order to avoid dense signatures, but also should not be too high in order to still perform a good fit of the observed counts.

Besides these parameters, we also need to estimate the initial values of beta to be used during the estimation. We now show how to do this on the set of counts from 560 tumors provided in Nik-Zainal, Serena, et al. (2016).
Extracting sparse mutational signatures via LASSO

Figure 1: Visualization of the counts from patient PD10010a from the dataset published in Nik-Zainal, Serena, et al
First, we can estimate the initial values of beta as follows.
Extracting sparse mutational signatures via LASSO

```r
starting.betas = starting.betas.estimation(x=patients,K=3:12,background_signature=background)
```

Then, we also need to explore the search space of values for the LASSO penalty in order to make a good choice. To do so, we can use the function `evaluate.lambda.range` to test different values as follows.

```r
lambda.range = evaluate.lambda.range(x=patients,K=10,beta=starting.betas[[8,1]],
                                       lambda_values=c(0.05,0.10))
```

As the executions of these functions can be very time-consuming, we also provide as examples together with the package a set of pre-computed results by the two functions `starting.betas.estimation` and `evaluate.lambda.range` obtained with the commands above.

```r
data(starting_betas_example)
data(lambda_range_example)
```

To evaluate the best lambda range, we need to carefully consider the log-likelihood of the solutions at each iteration of our method. This can be done by exploiting the `as.loglik.progression.in.range` functions that we provide. Here are some examples.

```r
# example of using too small a value of lambda
# the log-likelihood is very unstable across the iterations
res = as.loglik.progression.in.range(lambda.range.result=lambda_range_example,lambda_value=0.01)
plot(res)
```

![Example of using too small a value of lambda: the log-likelihood is very unstable across the iterations](image)

**Figure 2:** Example of using too small a value of lambda: the log-likelihood is very unstable across the iterations

```r
# example of using too high a value of lambda
# the log-likelihood drops after the first iteration
res = as.loglik.progression.in.range(lambda.range.result=lambda_range_example,lambda_value=0.30)
plot(res)
```
Extracting sparse mutational signatures via LASSO

\[ \text{Index} \]

\[ \text{res} \]

**Figure 3:** Example of using too high a value of lambda: the log-likelihood drops after the first iteration

```r
# example of using a good value of lambda
# the log-likelihood is increasing across the iterations
res <- as.loglik.progression.in.range(lambda.range.result=lambda_range_example,lambda_value=0.15)
plot(res)
```

**Figure 4:** Example of using a good value of lambda: the log-likelihood is increasing across the iterations

Now that we have evaluated all the required parameters, we need to decide which configuration of number of signatures and lambda value is the best. To do so, we rely on cross-validation.
cv = nmf.LassoCV(x=patients,K=3:10)

We notice that the computations for this task can be very time consuming, especially when many iterations of cross validations are specified (see manual) and a large set of configurations of the parameters are tested. To speed up the execution, we suggest using the parallel execution options. Also, to reduce the memory requirements, we advise splitting the cross validation in different runs, e.g., if one wants to perform 100 iterations, we would suggest making 10 independent runs of 10 iterations each. Also in this case, we provide as examples together with the package a set of pre-computed results obtained with the above command and the following settings: \( K = 3:10 \), cross validation entries = 0.10, lambda values = c(0.05, 0.10, 0.15), number of iterations of cross-validation = 2.

data(cv_example)

We can now estimate the best configuration of the parameters in terms of median mean squared error by cross validation, where the best configuration is the one with lowest error.

res = as.mean.squared.error(cv_example)$median
res_best = which(res==res[which.min(res)],arr.ind=TRUE)
best_K = rownames(res)[res_best[1]]
best_lambda = colnames(res)[res_best[2]]

best_K
## [1] "5_signatures"
best_lambda
## [1] "0.1_lambda"

Finally, we can compute the signatures for the best configuration, i.e., \( K = 5 \) and \( \lambda = 0.10 \).

beta = starting_betas_example["5_signatures","Value"]
res = nmf.LassoK(x=patients,K=5,beta=beta,background=background,lambda_rate=0.10,
iterations=5,num_processes=NA)

## Performing the discovery of the signatures by NMF with Lasso...
## Performing a total of 5 iterations...
## Progress 20%...
## Progress 40%...
## Progress 60%...
## Progress 80%...
## Progress 100%...

We conclude this vignette by plotting the discovered signatures.

signatures = as.beta(res)
signatures.plot(beta=signatures, xlabels=FALSE)

## Warning in melt(beta, varnames = c("signature", "cat")): The melt generic in
data.table has been passed a matrix and will attempt to redirect to the relevant
reshape2 method; please note that reshape2 is deprecated, and this redirection is
now deprecated as well. To continue using melt methods from reshape2 while both
libraries are attached, e.g. melt.list, you can prepend the namespace like reshape2::melt(beta).
In the next version, this warning will become an error.
Extracting sparse mutational signatures via LASSO

Figure 5: Visualization of the discovered signatures

4 sessionInfo()

- R version 3.6.1 (2019-07-05), x86_64-pc-linux-gnu
- Locale: LC_CTYPE=en_US.UTF-8, LC_NUMERIC=C, LC_TIME=en_US.UTF-8, LC_COLLATE=C, LC_MONETARY=en_US.UTF-8, LC_PAPER=en_US.UTF-8, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_US.UTF-8, LC_IDENTIFICATION=C
- Running under: Ubuntu 18.04.3 LTS
- Matrix products: default
- BLAS: /home/biocbuild/bbs-3.10-bioc/R/lib/libRblas.so
- LAPACK: /home/biocbuild/bbs-3.10-bioc/R/lib/libRlapack.so
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
Extracting sparse mutational signatures via LASSO

- Other packages: BSgenome 1.54.0, BSgenome.Hsapiens.1000genomes.hs37d5 0.99.1, Biobase 2.46.0, BiocGenerics 0.32.0, Biostrings 2.54.0, GenomInfoDb 1.22.0, GenomicRanges 1.38.0, IRanges 2.20.0, NMF 0.21.0, S4Vectors 0.24.0, SparseSignatures 1.6.0, XVector 0.26.0, bigmemory 4.5.33, cluster 2.1.0, knitr 1.25, pkgmaker 0.27, registry 0.5-1, rngtools 1.4, rtracklayer 1.46.0

- Loaded via a namespace (and not attached): BiocManager 1.30.9, BiocParallel 1.20.0, BiocStyle 2.14.0, DelayedArray 0.12.0, GenomInfoDbData 1.2.2, GenomicAlignments 1.22.0, Matrix 1.2-17, R6 2.4.0, RColorBrewer 1.1-2, RCurl 1.95-4.12, Rcpp 1.0.2, Rsamtools 2.2.0, SummarizedExperiment 1.16.0, XML 3.98-1.20, assertthat 0.2.1, bibtex 0.4.2, bigmemory.sri 0.1.3, bitops 1.0-6, codetools 0.2-16, colorspace 1.4-1, compiler 3.6.1, crayon 1.3.4, data.table 1.12.6, digest 0.6.22, doParallel 1.0.15, dplyr 0.8.3, evaluate 0.14, foreach 1.4.7, ggplot2 3.2.1, glue 1.3.1, grid 3.6.1, gridBase 0.4-7, gridExtra 2.3, gtable 0.3.0, highr 0.8, htmltools 0.4.0, iterators 1.0.12, labeling 0.3, lattice 0.20-38, lazyeval 0.2.2, magrittr 1.5, matrixStats 0.55.0, munsell 0.5.0, nnlasso 0.3, nnls 1.4, pillar 1.4.2, pkgconfig 2.0.3, plyr 1.8.4, purrr 0.3.3, reshape2 1.4.3, rlang 0.4.1, rmarkdown 1.16, scales 1.0.0, stringi 1.4.3, stringr 1.4.0, tibble 2.1.3, tidyselect 0.2.5, tools 3.6.1, withr 2.1.2, xfun 0.10, xtable 1.8-4, yaml 2.2.0, zlibbioc 1.32.0