SparseSignatures: An R package using LASSO-regularized non-negative matrix factorization to identify mutational signatures from human tumor samples

We outline the features of the R package SparseSignatures and its application to determine the signatures contributing to mutation profiles of tumor samples. We describe installation details and illustrate a step-by-step approach to (1) prepare the data for signature analysis, (2) determine the optimal parameters, and (3) employ them to determine the signatures and related exposure levels in the point mutation dataset.

Publisher’s note: Undertaking any experimental protocol requires adherence to local institutional guidelines for laboratory safety and ethics.
Protocol

SparseSignatures: An R package using LASSO-regularized non-negative matrix factorization to identify mutational signatures from human tumor samples

Lorenzo Mella,1 Avantika Lal,2 Fabrizio Angaroni,3 Davide Maspero,3 Rocco Piazza,4,5 Arend Sidow,6,8 Marco Antoniotti,3,5,8,* Alex Graudenzi,3,5,7,8,* and Daniele Ramazzotti3,8,9,10,*

1Department of Biomedical Sciences, Humanitas University, Rozzano, Italy
2Insitro, 279 E Grand Avenue, South San Francisco, CA, USA
3Department of Informatics, Systems and Communication, Università degli Studi di Milano-Bicocca, Milan, Italy
4Department of Medicine and Surgery, Università degli Studi di Milano-Bicocca, Milan, Italy
5Bicocca Bioinformatics Biostatistics and Bioimaging Centre – B4, Milan, Italy
6Department of Pathology, Stanford University, Stanford, CA, USA
7Institute of Molecular Bioimaging and Physiology, National Research Council (IBFM-CNR), Segrate, Italy
8Senior author
9Technical contact
10Lead contact
*Correspondence: marco.antoniotti@unimib.it (M.A.), alex.graudenzi@unimib.it (A.G.), daniele.ramazzotti@unimib.it (D.R.)
https://doi.org/10.1016/j.xpro.2022.101513

SUMMARY

We outline the features of the R package SparseSignatures and its application to determine the signatures contributing to mutation profiles of tumor samples. We describe installation details and illustrate a step-by-step approach to (1) prepare the data for signature analysis, (2) determine the optimal parameters, and (3) employ them to determine the signatures and related exposure levels in the point mutation dataset.

For complete details on the use and execution of this protocol, please refer to Lal et al. (2021).

BEFORE YOU BEGIN

Goals of the SparseSignatures package

The SparseSignatures package is designed to improve on previous decomposition methods of mutational signature discovery, applied to datasets of point mutations observed in cancer cells. The goal of the analysis is understanding and tracking processes involved in cancer initiation, decomposing mutation counts into meaningful frequency patterns (i.e., signatures), to simplify further etiological analysis.

The functionality provided by SparseSignatures is broadly summarized in the following two objectives:

1. Decompose the overall mutation counts of every individual patient as a combination of signatures and corresponding exposure levels.
2. Limit the effect of overfitting on the sample data, by encouraging a choice of signatures with greater sparsity, which may lead to an easier attribution of the discovered signatures to mutagenic processes.
Discovering signatures with a machine learning approach

Mutational signatures are discovered by analyzing ensemble point-mutation counts from a sample of individuals. Each individual record lists the count of mutations detected in the cancer sample, classified with respect to the trinucleotide contexts: the observed nucleotide substitution (C>A, C>G, C>T, T>A, T>C, or T>G) and the neighboring 5' and 3' nucleotides, for a total of 96 trinucleotide mutation types (Lal et al., 2021). The counts are naturally summarized in a rectangular data-frame (an R dataframe, in our case) whose row records refer to the individual cancer samples and whose columns correspond to the 96 types.

The SparseSignatures package takes inspiration from the non-negative matrix factorization (NMF), an unsupervised machine learning technique, which iteratively decomposes the point mutation count matrix into the desired number of constituent signatures, while simultaneously computing the exposure levels for each combination of signature and mutation type.

This technique aims at discovering a finite number of signatures. The goal is to provide for each signature a likely etiology and possibly linking it to prognostic factors. Signature databases have been compiled (see the COSMIC version 3 database at https://cancer.sanger.ac.uk/signatures, for example).

The NMF method has proved to be effective, as meaningful signatures have been identified in the literature. However, the algorithm is susceptible to discovering additional spurious signatures, that are not reflecting real prognostic factors, instead being artifacts that result from data overfitting (Lal et al., 2021). For example, a subset of the mutation counts that should be classified as the same signature may be split into multiple ones.

Formulation

Classical non-negative matrix factorization

Calling \( X \), the matrix representation of the sample point-mutation counts (and denoting by \( x_{ij} \) its entry at record \( i \) and mutation type \( j \) ), NMF aims at finding two matrices \( a \) and \( b \) such that the following decomposition holds, approximately:

\[
x_{ij} = a_{i0} b_{0j} + \sum_{k=1}^{K} a_{ik} \beta_{kj}.
\]

The same relations for every \( i \) and \( j \), can be compactly written in matrix form as:

\[
X = a_0 b_0 + a \beta.
\]

The interpretation of the involved matrices is as follows: the rows of \( \beta \) represent the discovered signatures, \( \beta_0 \) is a (optional) row vector representing the background signature (in our algorithm implementation, it is fixed during the learning step a priori), the columns of the matrix \( a \) represent the exposures of each sample in the dataset to each signature, and the (optional) column vector \( a_0 \) represents the exposures (also learned by NMF) to the background signature.

In standard NMF implementations, the matrices are found by iteratively updating the values of the tuneable matrix entries to minimize a discrepancy measure, such as the sum of squared errors:

\[
\|X - a_0 b_0 - a \beta\|_F^2 = \sum_{i=1}^{N} \sum_{j=1}^{96} (x_{ij} - a_{i0} b_{0j} - \sum_{k=1}^{K} a_{ik} \beta_{kj})^2.
\]

The minimization is subject to the non-negativity constraints \( a_0 \geq 0, a \geq 0, \) and \( \beta \geq 0 \).

SparseSignatures: NMF with sparsity: LASSO regularization

SparseSignatures performs an analogous minimization process, i.e., using a squared-error cost, subject to the same non-negativity constraints. However, it introduces an additional regularization term on the entries matrix \( \beta \), yielding a modified cost function of the form:
\[ \| X - a_0b_0 - a\beta \|_F^2 + \lambda_\beta \| \beta \|_1 = \sum_{i=1}^{N} \sum_{j=1}^{M} |x_{ij} - a_{i0}b_{0j} - \sum_{k=1}^{K} a_{ik}b_{kj}|^2 + \lambda_\beta \sum_{k=1}^{K} \sum_{j=1}^{M} |\beta_{kj}|. \]

The kind of regularization employed here is LASSO (or L1-regularization, due to the use of the L1-norm \( \| \cdot \|_1 \)). In regression and other machine learning contexts, LASSO regularization has the effect of increasing sparsity (i.e., the presence of entries close to zero) in the learned representation.

The regularization effect, dictating the level of sparsity, is tuned using the \( \lambda_\beta \) parameter. A value of 0 corresponds to no regularization – hence, the classical NMF method fixing a background signature – with higher values corresponding to higher and higher levels of sparsity in the resulting signatures (rows of \( \beta \)).

In principle, an L1-regularization term may also be added for the vector \( a_0 \) and the matrix \( a \), with the effect of “sparsifying” the exposure values. However, we will not discuss it in the present guide, as we have found that performing LASSO to the \( a \) values does not yield significantly improved results, while increasing the computational cost of the algorithm (Lal et al., 2021).

**Computational requirements**

The following setup is supported to run the workflow and perform the signature discovery analysis.

**Software**

3. Linux (kernel version 5.4 or 5.10: e.g., in Ubuntu 20.04 or above); Windows 11 or MacOS (High Sierra or above).
4. The R programming language (version >= 4.1) (https://www.r-project.org).
5. Broadband internet connection, during the installation phase.

Note that older versions of the mentioned operating systems are not supported, although the workflow is likely to run. In case of issues, verify potential incompatibilities on the individual SparseSignatures dependencies web pages.

**Hardware**

We tested the SparseSignatures package on the following architectures:

6. Intel® Core™ i7-4771 (3.5 GHz up to 3.9 GHz, 4 cores, 8 threads, 8 MB cache), with 16 GB DDR3 1600 MHz SDRAM.
7. Intel® Core™ i7-8550U (1.8 GHz up to 4.0 GHz, 4 cores, 8 threads, 8 MB cache), with 16 GB LPDDR3 2133 MHz SDRAM

The first architecture ran MacOS 10.13.6 High Sierra. The second architecture ran Linux (kernel ver. 5.10 LTS). As the multiprocessor dependencies on which SparseSignatures relies on are CPU-based, the user won’t need GPU hardware, whether integrated or discrete.

**Installation of BiocManager and SparseSignatures**

© Timing: 15 min

Several R packages needed to conduct the following analyses, including SparseSignatures itself, are not available on the standard R repositories (CRAN). Instead, they are provided by the bioinformatics-specific repository Bioconductor (https://bioconductor.org).

8. Install the Bioconductor Package Manager (BiocManager), running the snippet.
9. Install the SparseSignatures package:

```r
if (!require("BiocManager", quietly = TRUE)) {
  install.packages("BiocManager")
}

BiocManager::install("SparseSignatures")
```

**The NMF package (optional)**

© Timing: 2 min

The NMF R package (https://cran.r-project.org/package=NMF) provides multithreaded C++ implementations of the Non-negative Matrix Factorization algorithm. It enables shared memory among the worker processes and seamless memory mapping, in the case of very large matrices. It improves performance with large datasets and when validating optimal decomposition parameters.

Depending on the version, SparseSignatures may not install NMF extras alongside its own installation. If NMF extras are not installed when attaching SparseSignatures to the current working session, the user will be reminded of the optional NMF dependencies, and provided with simple instructions, which we reproduce here:

10. Install NMF extras, after loading the SparseSignatures package:

```r
install.extras("NMF")
```

NMF extras are not strictly required in case the system resources are adequate for the desired analysis. However, we will assume that they are installed.

**KEY RESOURCES TABLE**

| REAGENT or RESOURCE | SOURCE | IDENTIFIER |
|----------------------|--------|------------|
| Software and algorithms | Lal et al. (2021) | https://bioconductor.org/packages/SparseSignatures |
| MacOS or Windows or Ubuntu | Apple Inc. | https://www.apple.com |
| | Microsoft Corporation | https://www.microsoft.com |
| | Canonical Ltd | https://ubuntu.com |
| R | The R Foundation | https://www.r-project.org |
| Nnlasso | CRAN | Version 0.3 |
| Nnls | CRAN | Version 1.4 |
| data.table | CRAN | Version 1.14.2 |
| Biostrings | Bioconductor | Version 2.62.0 |
| GenomicRanges | Bioconductor | Version 1.46.1 |
| IRanges | Bioconductor | Version 2.28.0 |
| SGenome | Bioconductor | Version 1.62.0 |
| GenomeInfoDb | Bioconductor | Version 1.30.1 |

(Continued on next page)
STEP-BY-STEP METHOD DETAILS
Making the most of the provided code
We will break down the signature discovery procedure as a sequence of R snippets of code. They may be run interactively in the console provided by the R environment of choice, or composed in a .r script – optionally modified according to the user’s requirements. Notice that automating the whole procedure is infeasible, since the user’s direct interaction is required for the choice of the optimal parameters (see section “determining a valid range for the sparsity parameter”).

For illustration purposes, the protocol will make use of example datasets, derived from the breast-cancer mutation dataset analyzed in (Nik-Zainal et al., 2016). The user is invited to test the package using these datasets, and to inspect them to understand the required formats precisely, before using real data.

Input dataset specification

 © Timing: 1 h

The user must prepare a dataset detailing all the identified point mutations for all the patients observed in the study.

A typical starting point is given by one or more VCF (Variant Calling Format) files. For more information, a simple overview of the VCF file format can be found at: https://learn.gencore.bio.nyu.edu/ngs-file-formats/vcf-format.

1. Convert the dataset of interest into an R data.frame or matrix object. For example, a VCF file must be cleaned and saved in either a CSV or TSV file format as follows:
   a. Each record corresponds to a single mutation.
   b. The columns should have the following names, reporting the stated information:
      i. sample: the unique identifier of the patient/cancer sample from which the mutation is observed.
      ii. chrom: the affected chromosome.
      iii. start: the starting nucleotide of the affected sequence, as a numeric location referred to the chromosome.
      iv. end: the final nucleotide location (this should always be the same as the starting location for point-mutations).
      v. ref: the corresponding nucleotide in the reference genome.
      vi. alt: the altered nucleotide measured in the sample.

The exact DNA location allows retrieving the context of the mutation, which may be one out of 16 types, as explained above.
2. Import the constructed data file, in R using `read.delim` as follows:

```r
## If the values are comma-separated
mutations_df <- read.delim("/path/to/datafile.csv", sep = ",")
## OR
## If the values are TAB or whitespace-separated
mutations_df <- read.delim("/path/to/datafile.tsv", sep = "\t")
```

As an example of the features of the dataset, once loaded in memory, we load one of the example datasets from (Nik-Zainal et al., 2016), consisting of a list of point mutations in the DNA of breast cancer samples taken from three individuals (out of the 560 of the original whole dataset). The user can load it and inspect it as follows:

```r
data(ssm560_reduced)
```

Listing the first ten elements of the matrix, with

```r
head(ssm560_reduced, n = 10)
```

We obtain an example of the general form after which the `mutations_df` dataset should be modeled:

| sample | chrom | start   | end     | ref | alt |
|--------|-------|---------|---------|-----|-----|
| PD10014a | 1     | 1186484577 | 186484577 | A   | C   |
| PD10014a | 7     | 7141761948  | 141761948  | G   | A   |
| PD10014a | 7     | 71266228    | 71266228    | C   | T   |
| PD10014a | 8     | 82304475    | 82304475    | A   | T   |
| PD10014a | 3     | 191275626   | 191275626   | T   | A   |
| PD10014a | 4     | 4135265376  | 135265376   | C   | T   |
| PD10014a | 3     | 119344674   | 119344674   | G   | A   |
| PD10014a | 6     | 619019947   | 19019947    | G   | A   |
| PD10014a | 2     | 32318168    | 32318168    | C   | T   |
| PD10014a | 10    | 1389605097  | 89605097    | A   | T   |

### Generate the patient vs mutation matrix from mutation data

© Timing: 1–5 min

The `SparseSignatures` library provides facilities to effortlessly generate mutation frequency count data from point mutation data-frames. We will exemplify the procedure using the provided example datasets. The user can proceed with the `data.frame` prepared in the previous step.

3. Attach the `SparseSignatures` package. By doing this we will get seamless access to the functions `nmfLassoCV`, `nmfLasso`, etc., which we will employ below.
4. Install a reference human-genome specification from the Bioconductor repositories, within the package group BSgenome. The user must select, among the available choices, the reference genome consistent with the mutation dataset. In the case of the dataset illustrated above, the correct reference genome is hs37d5 (its information sheet can be downloaded from this FTP host). We continue with this choice, here, for the sake of illustration. We download the package BSgenome.Hsapiens.1000genomes.hs37d5, evaluating the snippet.

```r
if (!require("BSgenome.Hsapiens.1000genomes.hs37d5", quietly = TRUE)) {
  BiocManager::install("BSgenome.Hsapiens.1000genomes.hs37d5")
}
```

Different reference genomes can be adopted where needed, for instance to install hg38, one can use the following code:

```r
if (!require("BSgenome.Hsapiens.UCSC.hg38", quietly = TRUE)) {
  BiocManager::install("BSgenome.Hsapiens.UCSC.hg38")
}
```

5. Produce the trinucleotide-count matrix from the data-frame. This is automatically done using the function `import.trinucleotides.counts`:

```r
patients <- import.trinucleotides.counts(
  data = ssm560_reduced,
  reference = BSgenome.Hsapiens.1000genomes.hs37d5::hs37d5
)
```

When the user prints the matrix, the rows are named according to the three patient IDs in `ssm560_reduced`. The column names are expressed with an encoding specifying both the context and the base substitution. As an example, `A[C>A]G` represents a mutation `C>A` in the context `ApG`.

The timing information given at the beginning of the section is a rough estimate. The genome installation (7) should take little time depending on connection speeds. The main bottleneck is the `import.trinucleotides.counts` function. With a simulated dataset of 560 patients and 8000 mutations each, step (8) took about 80 s on our systems.

### Preparing a reference background signature

© Timing: 1 min

The background signature represents the residual mutation pattern invariably observed in most datasets, and that is usually explained by spontaneous occurrences of DNA mutation. While it can be omitted from the analysis, we suggest including it explicitly. The package provides two background signatures: the SBS55 signature extrapolated from the COSMIC database (Lal et al., 2021), and a signature derived from the human germline mutation spectrum (Rahbari et al., 2016).
6. Import one of the available background signatures:

```r
# To load the SBS5 signature from COSMIC
data(background)
# OR, for the human germline-derived signature
data(background2)
```

If in doubt, the user may adopt the SBS5 signature. As the user can verify, the two backgrounds are very similar according to several similarity measures and yield very similar results in practice. Both signatures have been slightly adjusted compared to the original sources. See (Lal et al., 2021) for details.

Determining the optimal signature number and sparsity to describe the data

This section deals with issues of technical nature relating to the sparsity parameter $\lambda_\beta$. We remind the user that although the SparseSignatures package allows to sparsify the exposure levels $\alpha$, this approach provided limited benefit while showing computational downsides. For this reason, we will set the sparsity parameter for the $\alpha$ values to zero in all the functions that require it. That is, only the signatures (beta) will be regularized and made sparse.

The package does not implement the signature parameter $\lambda_\beta$ in the same linear scale we used for simplicity in the theoretical presentation above. SparseSignatures allows for a fractional value instead, chosen between 0 and 1, with a value of 0 meaning no sparsification and a level of 1 meaning extreme sparsification.

Here we propose a systematic approach to the choice of the $\lambda_\beta$ parameter, using model selection with cross-validation.

Determining a valid range for the sparsity parameter

○ Timing: 4 min

Before we proceed with cross-validation, we need a heuristic approach to find levels of $\lambda_\beta$ that are within the working range of the algorithm: that is, values that ensure the convergence of the iterative procedure to meaningful outputs. To give an intuitive explanation of what could go wrong, imagine trying to fit many signatures, each with a high sparsity level. In such conditions, the random fluctuations pertaining to the dataset will have comparable scale to the excessively fragmented signatures, resulting in the signatures, and the exposures, being fundamentally misleading.

To find the appropriate working range, the user will have to manually test large values of $K$ (the expected maximum number of signatures) against the $\lambda_\beta$ values. The lambdaRangeBetaEvaluation function does this automatically.

7. Decide on a range for $K$. It should be large enough to realistically accommodate the correct number of signatures characterizing the dataset (e.g., from 3 to 10). The criteria for this decision are left to the user, as they are context dependent.

8. Choose a range of $\lambda_\beta$ percentage values, with roughly logarithmic spacing up to 0.2 (that is, 0.01, 0.02, 0.05, 0.1, 0.2) after which, proceed linearly (0.225, 0.25, 0.275, 0.3, ...).

9. Apply the function lambdaRangeBetaEvaluation using the ranges for both variables. The function simply runs NMF LASSO on all combinations and lists the results (for the description of the list elements, see the return value description of nmfLasso below).
lambdaRangeBetaEvaluation applies nmfLasso with the same specified arguments, except that it uses each value in lambda_test_values in turn. The variable results, therefore, holds a list of the corresponding nmfLasso outputs.

10. Inspect the results manually to verify whether there is a “cutoff” $\lambda_d$ value. If the loglik_progression entries appear to progressively decrease in absolute value, the combination of K and the corresponding lambda-value is feasible. If the entries quickly diverge (becoming more negative) and settle to a constant value after a few iterations, it suggests that the lambda value is infeasible.

```r
lambda_test_values <- c(0.01, 0.02, 0.05, 0.1, 0.2, 0.225, 0.25, 0.275, 0.3)
results <- lambdaRangeBetaEvaluation{
  x = patients,
  K = 5,
  lambda_values = lambda_test_values,
  background_signature = background,
  num_processes = 7
}
```

For instance, working on the example count matrix patients, we test how the $\lambda_d$ values listed in point 9 change in relation to a number of signatures fixed at K=11. Running the script in point 10, we can easily spot the breakdown threshold. Below we show the output only for values surrounding the threshold.

The log-likelihood progressions up to a value $\lambda_d=0.2$ of the sparsity parameter exemplify the expected behavior: a quick rise after the first value, and a stabilization in about 30 iterations. As soon as we hit the slightly higher choice of parameter $\lambda_d=0.225$, we observe, instead, a rapid divergence towards a fixed low log-likelihood value, which is evidence of lack of convergence.

```r
for (i in 1:length(lambda_test_values)){
  print(colnames(results)[[i]])
  print(results[[i]]$loglik_progression)
}
```
11. Select the values of $K$ and $\lambda_B$ to carry over to the cross-validation process. Bi-cross-validation tests all the combinations ($K$, $\lambda_B$) built from the chosen lists of values of the two parameters. The list of values of $K$ is the one tested above. The values of $\lambda_B$ are the ones used in $\text{lambdaRateBetaEvaluation}$, up to the highest that guarantees convergence for all $K$ (i.e., the largest just below all observed cutoffs).

A full description of the function $\text{lambdaRateBetaEvaluation}$ can be found in Table 1.

Determining the optimal signature number and sparsity by cross-validation

© Timing: 1 h per repetition

We are now in position to use cross-validation to determine the optimal values among user-supplied ranges for the number of signatures $K$ and the sparsity parameter $\lambda_B$.

12. Perform (bi-)cross-validation across the ranges for $K$ and for $\lambda_B$ (expressed in fractional scale) using the selected background signature and disabling regularization for the exposures $a$:

```r
cv_out <- nmfLassoCV(
  x = patients,
  K = 3:6,
  background_signature = background,
)```
The return value in `cv_out` is a list, comprising goodness-of-fit measures for each cross-validation repetition, as specified in the `cross_validation_repetitions` argument. We are interested in the first element of the list – a 3-index array specifying the mean squared errors (MSE) for all combinations of `lambda_values_alpha`, `lambda_values_beta` and `K` (in this order). The first index can be ignored: it comprises a single value for `lambda_values_alpha`, which we originally set to 0. Using the MSEs along the other two dimensions, we can determine their best combination by adopting a minimal MSE criterion.

13. Analyze the mean squared error results averaging over cross-validation repetitions. Within a single run, with multiple repetitions (`cross_validation_repetitions > 1`), obtain the mean MSEs for all combinations of `K` and `lambda_values_beta`:

```r
cv_mses <- cv_out$grid_search_mse[1, , ]
cv_means_mse <- matrix(
  sapply(cv_mses, FUN = mean),
  nrow = dim(cv_mses)[1]
)
dimnames(cv_means_mse) <- dimnames(cv_mses)
```

Compute the combination with the lowest MSE:

```r
min_ii <- which(cv_means_mse == min(cv_means_mse), arr.ind = TRUE)
min_Lambda <- rownames(cv_means_mse)[min_ii[1]]
min_K <- colnames(cv_means_mse)[min_ii[2]]
cat("Minimum MSE at:", min_Lambda, "and", min_K, "\n")
```

The output will look like the following:

```
Minimum MSE at: 0.01_Lambda_Beta and 5_Signatures
```

A full specification of `nmfLassoCV`, together with its arguments, is found in Table 2 below.

**Discovering the signatures within the dataset**

© Timing: 5 min
Having determined the optimal number of signatures $K$ and sparsity parameter $\lambda_\beta$ completes the preparatory steps for the analysis.

Thereafter, the results can be used to set up the signature discovery process, using the `nmfLasso` function, whose full argument specification is found in Table 3.

14. **Apply the `nmfLasso` function on the dataset (e.g., `patients`), specifying the values of $K$ and `lambda_rate_beta` found by cross-validation.** The default number of iterations, 30, should suffice in most cases. Since this step entails only one run of `nmfLasso`, the number of iterations can be raised for more stable results (e.g., 50). Keep the `lambda_rate_alpha` parameter to 0, to disable the regularization of the exposure values.

```r
nmf_lasso_out <- SparseSignatures::nmfLasso(
  x = patients,
  K = 4,
  background_signature = background,
  lambda_rate_alpha = 0,
  lambda_rate_beta = lambda_rate_beta_est,
  verbose = TRUE, log_file = "")
```
15. Analyze the output list `nmf_lasso_out`, whose information can be extracted using the named entries. These are summarized in the table below (N stands for the number of samples):

**Signature visualization and analysis**

© Timing: 1 min

After the analysis with `nmfLasso` has been performed, the user can read the signatures in the following ways:

16. Perform direct signature evaluation by considering the returned beta matrix:

```r
# For an individual (non-background) signature (2 <= k <= K)
mnf_lasso_out$beta[k,]
# For the exposure vector (size K - 1) of the n-th patient to the discovered signatures:
mnf_lasso_out$alpha[n, -1]
# For the exposure to the background signature:
mnf_lasso_out$alpha[n, 1]
```

17. Represent a number of signatures graphically with the `signatures.plot` function:

```r
beta <- nmf_lasso_out$beta
# Visualize all signatures (for up to 8 signatures)
signatures.plot(beta)
# Visualize a subset (for best results with more than 8 signatures)
signatures.plot(beta[c(2, 3, 5),])
```

**EXPECTED OUTCOMES**

The expected outcome of the signatures analysis is a set of estimated mutational signatures describing mutational processes active in the analyzed dataset, and the related exposure to samples. An example of visualization of mutational signatures is shown in Figure 1.

**LIMITATIONS**

`SparseSignatures` suffers from the expected limitations common to machine learning algorithms applied to high-dimensional data. It requires sizable datasets (i.e., at least hundreds of samples, also depending on the complexity/number of mutational signatures present in the dataset) for the signature discovery process to converge to meaningful results. As an example of a reasonable sample
We analyzed mutations from 560 breast-cancer patients (Nik-Zainal et al., 2016) and a dataset of 147 patients, comprising pancreatic cancer whole genomes (Lal et al., 2021). Further, it requires a grid search to choose the best values of $K$, $\lambda_\beta$ and optionally $\lambda_\alpha$ as well. It may become computationally infeasible for the user to test all desired combinations of these parameters, leading to failure to find the optimal setting.

Additionally, SparseSignatures is not robust to unbalanced datasets, that is, datasets where the number of mutations varies by orders of magnitude per cancer type. Since the signatures are discovered essentially as a partition of the original mutation counts, very low counts per individual patient (e.g., less than 1000) would reduce the confidence in the estimate of the signature entries. In this situation, the signatures returned by the algorithm would yield unreliable interpretations.

| Argument name       | Type                                      | Description                                                                                                                                 |
|---------------------|-------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| $X$                 | an $N \times 96$ matrix ($N$ number of samples) | The mutation count matrix.                                                                                                                                 |
| $K$                 | an integer vector                          | The candidate numbers of signatures whose goodness of fit is tested by cross-validation. Each entry of $K$ includes the background signature. |
| starting_beta       | a list of matrices                         | A list of matrices of dimensions $N \times k$, where $k$ ranges among the values given in the argument $K$. The matrices serve as initial values for the signature matrix $\beta$ and must include the background signature as the first row. |
| background_signature| a numeric vector of length 96             | A background signature provided by the user. The parameter is ignored if $\beta$ is given instead. If NULL, it is estimated through NMF.          |
| normalize_counts    | a Boolean                                  | If TRUE normalize the count matrix $X$ row-wise before processing it. Useful for algorithm stability, when considerably different total counts of mutations are observed among the patients. |
| nmf_runs            | an integer                                 | Number of iterations to estimate the $length(K)$ matrices $\beta$ (including the background signature) in case the argument $\beta$ is NULL. Ignored if $\beta$ is given. |
| lambda_values_alpha | a vector of doubles                        | The candidate values of the sparsity parameter for the exposure-matrix entries $a$ whose goodness of fit is assessed by cross-validation.       |
| lambda_values_beta  | a vector of doubles                        | The candidate values of the sparsity parameter for the signature matrix $\beta$ whose goodness of fit is assessed by cross-validation.       |
| cross_validation_entries | a double                              | The cross-validation test size, i.e., the percentage of entries set to zero during NMF and used for validation.                           |
| cross_validation_iterations | an integer                               | The number of randomized restarts of a single cross-validation repetition, in case of poor fits.                                      |
| cross_validation_repetitions | an integer                               | The number of repetitions of the cross-validation procedure.                                                                          |
| iterations          | an integer                                 | The number of iterations of every single run of NMF LASSO.                                                                           |
| max_iterations_lasso | an integer                                 | The number of sub-iterations involved in the sparsification phase, within a full NMF LASSO iteration.                                  |
| num_processes       | an integer or Inf                          | The number of requested NMF worker subprocesses to spawn. If Inf, an adaptive maximum number is automatically chosen. If NA or NULL, the function is run by the main R session as a single process. |
| seed                | an integer                                 | Seed for the random number generation. To be set for reproducibility.                                                                |
| verbose             | a Boolean                                  | If TRUE, informative messages are printed on the R console over the execution.                                                          |
| log_file            | a string                                   | The path to the text file where to record the console logs if parallel processes are used. Ignored if set to the empty string or if num_processes is disabled. |
**TROUBLESHOOTING**

**Problem 1**
Install R version 4.1 on Ubuntu 20.04 LTS.

We recommend using SparseSignatures with R version 4.1 or above. However, the Ubuntu 20.04 LTS repositories only provide R version 3.6. The upcoming Ubuntu 22.04 LTS release (expected May 2022 at the time of writing) is likely to include R version 4.1.2. In any case, depending on your Ubuntu release, you may have to follow the instructions below to install a version of R greater or equal to 4.1.

---

**Table 3. Specification of the function nmfLasso**

```r
nmfLasso(x, K, beta = NULL, background_signature = NULL, normalize_counts = TRUE, nmf_runs = 10,
         lambda_rate_alpha = 0.05, lambda_rate_beta = 0.05,
         iterations = 30, max_iterations_lasso = 10000,
         seed = NULL, verbose = TRUE)
```

| Argument name   | Type                     | Description                                                                 |
|-----------------|--------------------------|-----------------------------------------------------------------------------|
| `x`             | an \(N \times 96\) matrix (\(N\) samples) | The mutation count matrix                                                   |
| `K`             | an integer               | The number of signatures (min. value = 2) to be fit to the dataset, including the background signature. |
| `beta`          | an \(N \times K\) matrix | The initial value of the signature matrix \(\beta\). If NULL, it is estimated with a few runs of NMF. It must include the background signature as the first row. |
| `background_signature` | a numeric vector of length 96 | A background signature provided by the user. The parameter is ignored if `beta` is given instead. If NULL, it is estimated through NMF. |
| `normalize_counts` | a Boolean                | If TRUE normalize the count matrix `x` row-wise before processing it. Useful for algorithm stability, when considerably different total counts of mutations are observed among the patients. |
| `nmf_runs`      | an integer               | The number of runs of NMF used for `beta` or `background_signature` estimation, if required. |
| `lambda_rate_alpha` | a double between 0.0 and 1.0, inclusive | The sparsity parameter for the exposure values `alpha` (including the background-signature exposures). |
| `lambda_rate_beta` | a double between 0.0 and 1.0, inclusive | The sparsity parameter for the non-background signature entries `beta`. |
| `iterations`    | an integer               | The number of iterations in a single NMF LASSO algorithm run. |
| `max_iterations_lasso` | an integer or Inf       | The number of sub-iterations involved in the sparsification phase, within a full NMF LASSO iteration. |
| `seed`          | an integer               | Seed for the random number generation. To be set for reproducibility. |
| `verbose`       | a Boolean                | If TRUE, print completion-status and descriptive messages over the function run. |

**Label**

| Label         | Type                     | Description                                                                 |
|---------------|--------------------------|----------------------------------------------------------------------------|
| **Alpha**     | an \(N \times K\) matrix | It denotes the exposure vectors to the elements of each signature. The first column includes the exposures of the background signature. The other columns (labeled as S1, S2, etc.) refer to the discovered signatures. |
| **Beta**      | a \(K \times 96\) matrix | The rows are the discovered signatures, whose entries are labeled via column names with the explicit mutation-type encoding. The topmost row equals the provided background signature, passed through the `background_signature` argument. |
| **starting_alpha** | an \(N \times K\) matrix | The initial values of \(\alpha\) used at the start of the NMF-LASSO algorithm. Useful for troubleshooting purposes. |
| **starting_beta** | a \(K \times 96\) matrix | The initial values of \(\beta\) used at the start of the NMF-LASSO algorithm. Used for troubleshooting purposes. If the optional argument `beta` is used, this is the same object. |
| **loglik_progression** | a vector of floats | Its values offer a record of the log-likelihood measured at every iteration of the algorithm. The expected behavior is a sharp drop at the first few iterations and a smooth decrease thereafter. |
| **best_loglik** | a float                  | The lowest value of the log-likelihood obtained over the training iterations. This is the value at which the final \(\alpha\) and \(\beta\) are chosen. |
Potential solution

To install R, version 4.1, on Ubuntu 20.04 LTS, the user needs to enable the CRAN PPA as an installation source. We closely follow the procedure illustrated on the CRAN website (https://cran.r-project.org).

Warning: these steps will replace any previous version of R (e.g., version 3.6) with version 4.1.

In a terminal window, update the current repositories and make sure that the packages gnupg and dirmngr are installed:

```
sudo apt update
sudo apt install gnupg dirmngr
```

Next, download an official CRAN public key and add it to the Ubuntu keyring. This is necessary to verify the origin of packages installed from the PPA. Notice that the next two code-blocks are single lines, wrapped here due to page constraints:

```
wget -qO- https://cloud.r-project.org/bin/linux/ubuntu/
Marutter_pubkey.asc |
sudo tee -a /etc/apt/trusted.gpg.d/cran_ubuntu_key.asc
```

Finally, add the PPA to the list of apt sources:

```
sudo add-apt-repository *deb https://cloud.r-project.org/bin/linux/ubuntu $(lsb_release -cs) -cran40/*
```
It is now possible to install R version 4.1:

```
apt install -no-install-recommends r-base
```

**Problem 2**
Failure while installing SparseSignatures on Ubuntu 20.04 LTS.

The installation of SparseSignatures from Bioconductor, on a clean installation of Ubuntu 20.04 LTS, may end in failure if the system does not provide the required C headers for the compilation of selected dependencies.

**Potential solution**
The only additional dependencies that are required after the installation of R can be installed issuing the following command in a terminal:

```
sudo apt install libxml2-dev libcurl4-gnutls-dev
```

After the brief installation, carry out the instructions to install SparseSignatures again, to complete the process.

**Problem 3**
Execution of nmfLassoCV halts or is very slow.

The function nmfLassoCV is the most resource-heavy of the package. Since a higher number of CV repetitions corresponds to more accurate parameter estimates, it is desirable to set the `cross_validation_repetitions` argument to a high value. However, with the suggested configuration, CV repetitions are carried out by individual processes, which must share time on the available processors (i.e., CPU cores). Moreover, the higher the number of processes the higher the consumption of memory: with our example dataset, each process will require about 900 MiB.

The time estimates that we provided at the top of the relevant section scale linearly with the `cross_validation_repetitions` parameter. However, if the user is experiencing significantly longer times, it is possible that the system cannot handle the computational load memory-wise. In this case, we suggest carrying out cross-validation by running `nmfLassoCV` multiple times, saving the results and manually averaging them at the end. The procedure yields essentially identical results, simply replacing with a short script what the function would do automatically.

Decide on a split of the overall repetitions as a product of the number `n_outer_reps` of times that `nmfLassoCV` is run, and the value `n_inner_reps` to assign to the argument `cross_validation_repetitions` in each `nmfLassoCV` call. For example, to perform cross-validation with the equivalent of 50 repetitions, you may set

```
n_inner_reps <- 5
n_outer_reps <- 10
```

Keep in mind that it is better to have low values of `n_inner_reps`, to avoid the computational issues discussed above. High values of `n_outer_reps` are not critical in this regard.

Carry out multiple cross-validation runs and collect the results into a list:

```
cv_out_list <- list()
for (i in 1:n_outer_reps) {
  ...
}
```
Extract the MSE data from the elements of `cv_out_list` and construct a matrix of sums of the MSE values, classified by the \( K \) and \( \lambda_\beta \) values under consideration:

```r
cv_mses_list <- lapply(
    cv_out_list,
    FUN = function (x) x$grid_search_mse[1, , ]
)

sum_repetitions <- function(mses) {
    mse_sums <- matrix(
        sapply(mses, FUN = sum),
        nrow = dim(mses)[1]
    )
    dimnames(mse_sums) <- dimnames(mses)
    return(mse_sums)
}

cv_sum_mses_list <- lapply(cv_mses_list, FUN = sum_repetitions)

cv_means_mse <-
    reduce('+', cv_sum_mses_list) / (n_inner_reps * n_outer_reps)
```

Use the resulting matrix to find the combination of parameters that yields the lowest MSE (this step is the same as the normal procedure):

```r
min_ii = which(cv_means_mse == min(cv_means_mse), arr.ind = TRUE)
min_Lambda = rownames(cv_means_mse)[min_ii[1]]
min_K = colnames(cv_means_mse)[min_ii[2]]
cat("Minimum MSE at: ", min_Lambda, " and ", min_K, "\n")
```
Problem 4
Uncertain estimation of the number of signatures

The estimation of the optimal number of signatures $K$ is uncertain, as the mean squared error does not show a clear trend and multiple solutions seem to be similarly good.

Potential solution
Perform a higher number of cross validation iterations until the results are more clear.

Problem 5
Few signatures are discovered

The optimal number of signatures is low and expected mutational processes might be missed.

Potential solution
The inference task can be difficult when combining multiple datasets obtained with different technologies. In this case, the cross-validation estimate might be conservative and return only the most significant signatures, common to most samples in the dataset. One may wish to repeat the analysis on different subsets of the dataset, if there is clear evidence of poor signatures/samples fit.

Warning: one must pay attention when approaching the problem this way, as results will likely be affected by overfitting.

RESOURCE AVAILABILITY

Lead contact
Further information and requests for resources should be directed to and will be fulfilled by the lead contact, Daniele Ramazzotti (daniele.ramazzotti@unimib.it).

Materials availability
The official Bioconductor page of the package, which includes the Reference Manual is available at https://bioconductor.org/packages/SparseSignatures.

Data and code availability
The source code for the SparseSignatures package, together with the example data-files, is freely available on GitHub at https://github.com/danro9685/SparseSignatures under Apache Licence 2.0. The software was deposited on Zenodo at https://doi.org/10.5281/zenodo.6619223.

ACKNOWLEDGMENTS

We acknowledge that this research was partially supported by the Italian Ministry of University and Research (MIUR) - Department of Excellence project PREMIA (PREcision Medicine Approach: bringing biomarker research to the clinic). This work is partially supported by a Bicocca 2020 Starting Grant to F.A. and D.R.

AUTHOR CONTRIBUTIONS
Writing – original draft and writing – review & editing, L.M., A.L., M.A., A.G., and D.R.; software, L.M., D.M., A.L., and D.R.; funding acquisition, F.A., R.P., M.A., A.G., and D.R.; supervision, A.S., M.A., A.G., and D.R. All authors read, revised, and approved the manuscript.

DECLARATION OF INTERESTS
A.L. is an employee of Insitro, South San Francisco, CA, USA. Insitro had no involvement in this work.
REFERENCES

Lal, A., Liu, K., Tibshirani, R., Sidow, A., and Ramazzotti, D. (2021). De novo mutational signature discovery in tumor genomes using SparseSignatures. PLoS Comput. Biol. 17, e1009119. https://doi.org/10.1371/journal.pcbi.1009119.

Nik-Zainal, S., Davies, H., Staal, J., Ramakrishna, M., Glodzik, D., Zou, X., Martincorena, I., Alexandrov, L.B., Martin, S., Wedge, D.C., and Van Loo, P. (2016). Landscape of somatic mutations in 560 breast cancer whole-genome sequences. Nature 534, 47–54.

Rahbari, R., Wuster, A., Lindsay, S.J., Hardwick, R.J., Alexandrov, L.B., Turki, S.A., Dominiczak, A., Morris, A., Porteous, D., Smith, B., et al. (2016). Timing, rates and spectra of human germline mutation. Nat. Genet. 48, 126–133.