An introduction to NMF package

Version 0.23.6

Renaud Gaujoux

March 28, 2020

This vignette presents the *NMF* package[1] (Gaujoux et al. 2010), which implements a framework for Nonnegative Matrix Factorization (NMF) algorithms in R (R Development Core Team 2011). The objective is to provide an implementation of some standard algorithms, while allowing the user to easily implement new methods that integrate into a common framework, which facilitates analysis, result visualisation or performance benchmarking. If you use the package *NMF* package in your analysis and publications please cite:

Renaud Gaujoux et al. "A flexible R package for nonnegative matrix factorization". In: *BMC Bioinformatics* 11.1 (2010), p. 367. ISSN: 1471-2105. DOI: 10.1186/1471-2105-11-367

Note that the *NMF* package includes several NMF algorithms, published by different authors. Please make sure to also cite the paper(s) associated with the algorithm(s) you used. Citations for those can be found in Table 1 and in the dedicated help pages `?gedAlgorithm.<algorithm>`, e.g., `?gedAlgorithm.SNMF_R`.

Installation: The latest stable version of the package can be installed from any CRAN repository mirror:

```
# Install
install.packages('NMF')
# Load
library(NMF)
```

The *NMF* package is a project hosted on *R-forge*[2]. The latest development version is available from https://r-forge.r-project.org/R/?group_id=649 and may be installed from there[3].

Support: UseRs interested in this package are encouraged to subscribe to the user mailing list (nmf-user@lists.r-forge.r-project.org), which is the preferred channel for enquiries, bug reports, feature requests, suggestions or NMF-related discussions. This will enable better tracking as well as fruitful community exchange.

Important: Note that some of the classes defined in the NMF package have gained new slots. If you need to load objects saved in versions prior 0.8.14 please use:

```
# eg., load from some RData file
load('object.RData')
# update class definition
object <- nmfObject(object)
```

[1] https://cran.r-project.org/package=NMF
[2] https://r-forge.r-project.org/projects/nmf
[3] `install.packages("NMF", repos = "http://R-Forge.R-project.org")`

Contents

1 Overview **3**
- 1.1 Package features . . . 3
- 1.2 Nonnegative Matrix Factorization . . . 3
- 1.3 Algorithms . . . 4
- 1.4 Initialization: seeding methods . . . 4
- 1.5 How to run NMF algorithms . . . 6
- 1.6 Performances . . . 7
- 1.7 How to cite the package NMF . . . 8

2 Use case: Golub dataset **8**
- 2.1 Single run . . . 9
  - 2.1.1 Performing a single run . . . 9
  - 2.1.2 Handling the result . . . 9
  - 2.1.3 Extracting metagene-specific features . . . 11
- 2.2 Specifying the algorithm . . . 11
  - 2.2.1 Built-in algorithms . . . 11
  - 2.2.2 Custom algorithms . . . 13
- 2.3 Specifying the seeding method . . . 13
  - 2.3.1 Built-in seeding methods . . . 13
  - 2.3.2 Numerical seed . . . 13
  - 2.3.3 Fixed factorization . . . 14
  - 2.3.4 Custom function . . . 14
- 2.4 Multiple runs . . . 15
- 2.5 Parallel computations . . . 16
  - 2.5.1 Memory considerations . . . 16
  - 2.5.2 Parallel foreach backends . . . 16
  - 2.5.3 Runtime options . . . 16
  - 2.5.4 High Performance Computing on a cluster . . . 17
  - 2.5.5 Forcing sequential execution . . . 18
- 2.6 Estimating the factorization rank . . . 20
  - 2.6.1 Overfitting . . . 22
- 2.7 Comparing algorithms . . . 22
- 2.8 Visualization methods . . . 23

3 Extending the package **24**
- 3.1 Custom algorithm . . . 25
  - 3.1.1 Using a custom algorithm . . . 25
  - 3.1.2 Using a custom distance measure . . . 26
  - 3.1.3 Defining algorithms for mixed sign data . . . 27
  - 3.1.4 Specifying the NMF model . . . 28
- 3.2 Custom seeding method . . . 30

4 Advanced usage **31**
- 4.1 Package specific options . . . 31

5 Session Info **33**

References **33**

1 Overview

1.1 Package features

This section provides a quick overview of the *NMF* package package's features. Section 2 provides more details, as well as sample code on how to actually perform common tasks in NMF analysis.

The *NMF* package package provides:

- 11 built-in algorithms;
- 4 built-in seeding methods;
- Single interface to perform all algorithms, and combine them with the seeding methods;
- Provides a common framework to test, compare and develop NMF methods;
- Accept custom algorithms and seeding methods;
- Plotting utility functions to visualize and help in the interpretation of the results;
- Transparent parallel computations;
- Optimized and memory efficient C++ implementations of the standard algorithms;
- Optional layer for bioinformatics using BioConductor (Gentleman et al. 2004);

1.2 Nonnegative Matrix Factorization

This section gives a formal definition for Nonnegative Matrix Factorization problems, and defines the notations used throughout the vignette.

Let X be a $n \times p$ non-negative matrix, (i.e with $x_{ij} \geq 0$, denoted $X \geq 0$), and $r > 0$ an integer. Non-negative Matrix Factorization (NMF) consists in finding an approximation

$$X \approx WH \ , \tag{1}$$

where W, H are $n \times r$ and $r \times p$ non-negative matrices, respectively. In practice, the factorization rank r is often chosen such that $r \ll \min(n, p)$. The objective behind this choice is to summarize and split the information contained in X into r factors: the columns of W.

Depending on the application field, these factors are given different names: basis images, metagenes, source signals. In this vignette we equivalently and alternatively use the terms *basis matrix* or *metagenes* to refer to matrix W, and *mixture coefficient matrix* and *metagene expression profiles* to refer to matrix H.

The main approach to NMF is to estimate matrices W and H as a local minimum:

$$\min_{W,H \geq 0} \underbrace{[D(X, WH) + R(W, H)]}_{=F(W,H)} \tag{2}$$

where

- D is a loss function that measures the quality of the approximation. Common loss functions are based on either the Frobenius distance

$$D : A, B \mapsto \frac{Tr(AB^t)}{2} = \frac{1}{2} \sum_{ij} (a_{ij} - b_{ij})^2,$$

  or the Kullback-Leibler divergence.

$$D : A, B \mapsto KL(A||B) = \sum_{i,j} a_{ij} \log \frac{a_{ij}}{b_{ij}} - a_{ij} + b_{ij}.$$

- R is an optional regularization function, defined to enforce desirable properties on matrices W and H, such as smoothness or sparsity (Cichocki et al. 2008).

1.3 Algorithms

NMF algorithms generally solve problem (2) iteratively, by building a sequence of matrices (W_k, H_k) that reduces at each step the value of the objective function F. Beside some variations in the specification of F, they also differ in the optimization techniques that are used to compute the updates for (W_k, H_k).

For reviews on NMF algorithms see (Berry et al. 2007; Chu et al. 2004) and references therein.

The *NMF* package package implements a number of published algorithms, and provides a general framework to implement other ones. Table 1 gives a short description of each one of the built-in algorithms:

The built-in algorithms are listed or retrieved with function `nmfAlgorithm`. A given algorithm is retrieved by its name (a `character` key), that is partially matched against the list of available algorithms:

```
# list all available algorithms
nmfAlgorithm()

##  [1] "brunet"    "KL"        "lee"       "Frobenius" "offset"    "nsNMF"
##  [7] "ls-nmf"    "pe-nmf"    "siNMF"     "snmf/r"    "snmf/l"

# retrieve a specific algorithm: 'brunet'
nmfAlgorithm('brunet')

## <object of class: NMFStrategyIterative>
##  name: brunet [NMF]
##  objective: 'KL'
##  model: NMFstd
##  <Iterative schema>
##   onInit: none
##   Update: function (i, v, x, copy = FALSE, eps = .Machine$double.eps, ...)
##   Stop: 'connectivity'
##   onReturn: none

# partial match is also fine
identical(nmfAlgorithm('br'), nmfAlgorithm('brunet'))

## [1] TRUE
```

1.4 Initialization: seeding methods

NMF algorithms need to be initialized with a seed (i.e. a value for W_0 and/or H_0[4]), from which to start the iteration process. Because there is no global minimization algorithm, and due to the problem's high dimensionality, the choice of the initialization is in fact very important to ensure meaningful results.

The more common seeding method is to use a random starting point, where the entries of W and/or H are drawn from a uniform distribution, usually within the same range as the target matrix's entries. This method is very simple to implement. However, a drawback is that to achieve stability one has to perform multiple runs, each with a different starting point. This significantly increases the computation time needed to obtain the desired factorization.

To tackle this problem, some methods have been proposed so as to compute a reasonable starting point from the target matrix itself. Their objective is to produce deterministic algorithms that need to run only once, still giving meaningful results.

[4]Some algorithms only need one matrix factor (either W or H) to be initialized. See for example the SNMF/R(L) algorithm of Kim and Park (Kim et al. 2007).

Key	Description
`brunet`	Standard NMF. Based on Kullback-Leibler divergence, it uses simple multiplicative updates from (Lee et al. 2001), enhanced to avoid numerical underflow. $$H_{kj} \leftarrow H_{kj} \frac{\left(\sum_l \frac{W_{lk} V_{lj}}{(WH)_{lj}}\right)}{\sum_l W_{lk}} \quad (3)$$ $$W_{ik} \leftarrow W_{ik} \frac{\sum_l [H_{kl} A_{il}/(WH)_{il}]}{\sum_l H_{kl}} \quad (4)$$ **Reference:** (Brunet et al. 2004)
`lee`	Standard NMF. Based on euclidean distance, it uses simple multiplicative updates $$H_{kj} \leftarrow H_{kj} \frac{(W^T V)_{kj}}{(W^T W H)_{kj}} \quad (5)$$ $$W_{ik} \leftarrow W_{ik} \frac{(V H^T)_{ik}}{(W H H^T)_{ik}} \quad (6)$$ **Reference:** (Lee et al. 2001)
`nsNMF`	Non-smooth NMF. Uses a modified version of Lee and Seung's multiplicative updates for Kullback-Leibler divergence to fit a extension of the standard NMF model. It is meant to give sparser results. **Reference:** (Pascual-Montano et al. 2006)
`offset`	Uses a modified version of Lee and Seung's multiplicative updates for euclidean distance, to fit a NMF model that includes an intercept. **Reference:** (Badea 2008)
`pe-nmf`	Pattern-Expression NMF. Uses multiplicative updates to minimize an objective function based on the Euclidean distance and regularized for effective expression of patterns with basis vectors. **Reference:** (Zhang et al. 2008)
`snmf/r`, `snmf/l`	Alternating Least Square (ALS) approach. It is meant to be very fast compared to other approaches. **Reference:** (Kim et al. 2007)

Table 1: Description of the implemented NMF algorithms. The first column gives the key to use in the call to the `nmf` function.

For a review on some existing NMF initializations see (Albright et al. 2006) and references therein.

The *NMF* package package implements a number of already published seeding methods, and provides a general framework to implement other ones. Table 2 gives a short description of each one of the built-in seeding methods:

The built-in seeding methods are listed or retrieved with function `nmfSeed`. A given seeding method is retrieved by its name (a `character` key) that is partially matched against the list of available seeding methods:

```
# list all available seeding methods
nmfSeed()

## [1] "none"   "random" "ica"    "nndsvd"
```

```r
# retrieve a specific method: 'nndsvd'
nmfSeed('nndsvd')

## <object of class:  NMFSeed >
## name:  nndsvd
## method:  <function>

# partial match is also fine
identical(nmfSeed('nn'), nmfSeed('nndsvd'))

## [1] TRUE
```

Key	Description
`ica`	Uses the result of an Independent Component Analysis (ICA) (from the *fastICA* package[5] (Marchini et al. 2019)). Only the positive part of the result are used to initialize the factors.
`nnsvd`	Nonnegative Double Singular Value Decomposition. The basic algorithm contains no randomization and is based on two SVD processes, one approximating the data matrix, the other approximating positive sections of the resulting partial SVD factors utilizing an algebraic property of unit rank matrices. It is well suited to initialize NMF algorithms with sparse factors. Simple practical variants of the algorithm allows to generate dense factors. **Reference:** (Boutsidis et al. 2008)
`none`	Fix seed. This method allows the user to manually provide initial values for both matrix factors.
`random`	The entries of each factors are drawn from a uniform distribution over $[0, max(V)]$, where V is the target matrix.

Table 2: Description of the implemented seeding methods to initialize NMF algorithms. The first column gives the key to use in the call to the `nmf` function.

1.5 How to run NMF algorithms

Method `nmf` provides a single interface to run NMF algorithms. It can directly perform NMF on object of class `matrix` or `data.frame` and `ExpressionSet` – if the *Biobase* package[6] (Huber et al. 2015) is installed. The interface has four main parameters:

```r
nmf(x, rank, method, seed, ...)
```

`x` is the target `matrix`, `data.frame` or `ExpressionSet` [7]

`rank` is the factorization rank, i.e. the number of columns in matrix W.

`method` is the algorithm used to estimate the factorization. The default algorithm is given by the package specific option `'default.algorithm'`, which defaults to `'brunet'` on installation (Brunet et al. 2004).

`seed` is the seeding method used to compute the starting point. The default method is given by the package specific option `'default.seed'`, which defaults to `'random'` on initialization (see method `?rnmf` for details on its implementation).

See also `?nmf` for details on the interface and extra parameters.

[6] http://www.bioconductor.org/packages/release/bioc/html/Biobase.html

[7] `ExpressionSet` is the base class for handling microarray data in BioConductor, and is defined in the *Biobase* package.

1.6 Performances

Since version 0.4, some built-in algorithms are optimized in C++, which results in a significant speed-up and a more efficient memory management, especially on large scale data.

The older R versions of the concerned algorithms are still available, and accessible by adding the prefix `'.R#'` to the algorithms' access keys (e.g. the key `'.R#offset'` corresponds to the R implementation of NMF with offset (Badea 2008)). Moreover they do not show up in the listing returned by the `nmfAlgorithm` function, unless argument `all=TRUE`:

```
nmfAlgorithm(all=TRUE)

##  [1] ".R#brunet" "brunet"    "KL"        ".R#lee"    "lee"       "Frobenius"
##  [7] ".R#offset" "offset"    ".R#nsNMF"  "nsNMF"     ".M#brunet" ".ls-nmf"
## [13] "ls-nmf"    "pe-nmf"    ".siNMF"    "siNMF"     "snmf/r"    "snmf/l"

# to get all the algorithms that have a secondary R version
nmfAlgorithm(version='R')

##      brunet         lee      offset       nsNMF
## ".R#brunet"    ".R#lee" ".R#offset"  ".R#nsNMF"
```

Table 3 shows the speed-up achieved by the algorithms that benefit from the optimized code. All algorithms were run once with a factorization rank equal to 3, on the Golub data set which contains a 5000×38 gene expression matrix. The same numeric random seed (`seed=123456`) was used for all factorizations. The columns C and R show the elapsed time (in seconds) achieved by the C++ version and R version respectively. The column *Speed.up* contains the ratio R/C.

```
# retrieve all the methods that have a secondary R version
meth <- nmfAlgorithm(version='R')
meth <- c(names(meth), meth)
meth

##                                                    brunet         lee
##    "brunet"       "lee"    "offset"     "nsNMF" ".R#brunet"    ".R#lee"
##      offset       nsNMF
## ".R#offset"  ".R#nsNMF"

# load the Golub data
data(esGolub)

# compute NMF for each method
res <- nmf(esGolub, 3, meth, seed=123456)

## Compute NMF method 'brunet' [1/8] ... OK
## Compute NMF method 'lee' [2/8] ... OK
## Compute NMF method 'offset' [3/8] ... OK
## Compute NMF method 'nsNMF' [4/8] ... OK
## Compute NMF method '.R#brunet' [5/8] ... OK
## Compute NMF method '.R#lee' [6/8] ... OK
## Compute NMF method '.R#offset' [7/8] ... OK
## Compute NMF method '.R#nsNMF' [8/8] ... OK

# extract only the elapsed time
t <- sapply(res, runtime)[3,]
```

	C	R	Speed.up
brunet	1.93	11.15	5.79
lee	2.07	6.50	3.14
offset	4.64	8.42	1.81
nsNMF	3.11	16.95	5.45

Table 3: Performance speed up achieved by the optimized C++ implementation for some of the NMF algorithms.

1.7 How to cite the package NMF

To view all the package's bibtex citations, including all vignette(s) and manual(s):

```
# plain text
citation('NMF')

# or to get the bibtex entries
toBibtex(citation('NMF'))
```

2 Use case: Golub dataset

We illustrate the functionalities and the usage of the *NMF* package package on the – now standard – Golub dataset on leukemia. It was used in several papers on NMF (Brunet et al. 2004; Gao et al. 2005) and is included in the *NMF* package package's data, wrapped into an `ExpressionSet` object. For performance reason we use here only the first 200 genes. Therefore the results shown in the following are not meant to be biologically meaningful, but only illustrative:

```
data(esGolub)
esGolub

## ExpressionSet (storageMode: lockedEnvironment)
## assayData: 5000 features, 38 samples
##   element names: exprs
## protocolData: none
## phenoData
##   sampleNames: ALL_19769_B-cell ALL_23953_B-cell ... AML_7 (38 total)
##   varLabels: Sample ALL.AML Cell
##   varMetadata: labelDescription
## featureData
##   featureNames: M12759_at U46006_s_at ... D86976_at (5000 total)
##   fvarLabels: Description
##   fvarMetadata: labelDescription
## experimentData: use 'experimentData(object)'
## Annotation:

esGolub <- esGolub[1:200,]

# remove the uneeded variable 'Sample' from the phenotypic data
esGolub$Sample <- NULL
```

Note: To run this example, the `Biobase` package from BioConductor is required.

2.1 Single run

2.1.1 Performing a single run

To run the default NMF algorithm on data `esGolub` with a factorization rank of 3, we call:

```
# default NMF algorithm
res <- nmf(esGolub, 3)
```

Here we did not specify either the algorithm or the seeding method, so that the computation is done using the default algorithm and is seeded by the default seeding methods. These defaults are set in the package specific options `'default.algorithm'` and `'default.seed'` respectively.

See also Sections 2.2 and 2.3 for how to explicitly specify the algorithm and/or the seeding method.

2.1.2 Handling the result

The result of a single NMF run is an object of class `NMFfit`, that holds both the fitted NMF model and data about the run:

```
res

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 200
##   basis/rank: 3
##   samples: 38
##  # Details:
##   algorithm:  brunet
##   seed:  random
##   RNG: 10403L, 624L, ..., 449848215L [825a7345133c6f7869527d185527e323]
##   distance metric:  'KL'
##   residuals:  543535.7
##   Iterations: 510
##   Timing:
##      user  system elapsed
##     0.152   0.000   0.152
```

The fitted model can be retrieved via method `fit`, which returns an object of class `NMF`:

```
fit(res)

## <Object of class:NMFstd>
## features: 200
## basis/rank: 3
## samples: 38
```

The estimated target matrix can be retrieved via the generic method `fitted`, which returns a – generally big – `matrix`:

```
V.hat <- fitted(res)
dim(V.hat)

## [1] 200  38
```

Quality and performance measures about the factorization are computed by method `summary`:

```
summary(res)

##             rank sparseness.basis  sparseness.coef  silhouette.coef
##     3.000000e+00     6.392676e-01     6.217884e-01     8.126484e-01
## silhouette.basis        residuals            niter              cpu
##     7.487792e-01     5.435357e+05     5.100000e+02     1.520000e-01
##          cpu.all             nrun
##     1.520000e-01     1.000000e+00

# More quality measures are computed, if the target matrix is provided:
summary(res, target=esGolub)

##             rank sparseness.basis  sparseness.coef              rss
##     3.000000e+00     6.392676e-01     6.217884e-01     1.535504e+09
##             evar  silhouette.coef silhouette.basis        residuals
##     8.232656e-01     8.126484e-01     7.487792e-01     5.435357e+05
##            niter              cpu          cpu.all             nrun
##     5.100000e+02     1.520000e-01     1.520000e-01     1.000000e+00
```

If there is some prior knowledge of classes present in the data, some other measures about the unsupervised clustering's performance are computed (purity, entropy, ...). Here we use the phenotypic variable `Cell` found in the Golub dataset, that gives the samples' cell-types (it is a factor with levels: T-cell, B-cell or `NA`):

```
summary(res, class=esGolub$Cell)

##             rank sparseness.basis  sparseness.coef           purity
##     3.000000e+00     6.392676e-01     6.217884e-01     8.157895e-01
##          entropy  silhouette.coef silhouette.basis        residuals
##     3.926954e-01     8.126484e-01     7.487792e-01     5.435357e+05
##            niter              cpu          cpu.all             nrun
##     5.100000e+02     1.520000e-01     1.520000e-01     1.000000e+00
```

The basis matrix (i.e. matrix W or the metagenes) and the mixture coefficient matrix (i.e matrix H or the metagene expression profiles) are retrieved using methods `basis` and `coef` respectively:

```
# get matrix W
w <- basis(res)
dim(w)

## [1] 200   3

# get matrix H
h <- coef(res)
dim(h)

## [1]  3 38
```

If one wants to keep only part of the factorization, one can directly subset on the `NMF` object on features and samples (separately or simultaneously). The result is a `NMF` object composed of the selected rows and/or columns:

```
# keep only the first 10 features
res.subset <- res[1:10,]
class(res.subset)

## [1] "NMFfit"
## attr(,"package")
## [1] "NMF"

dim(res.subset)

## [1] 10 38  3

# keep only the first 10 samples
dim(res[,1:10])

## [1] 200  10   3

# subset both features and samples:
dim(res[1:20,1:10])

## [1] 20 10  3
```

2.1.3 Extracting metagene-specific features

In general NMF matrix factors are sparse, so that the metagenes can usually be characterized by a relatively small set of genes. Those are determined based on their relative contribution to each metagene.

Kim and Park (Kim et al. 2007) defined a procedure to extract the relevant genes for each metagene, based on a gene scoring schema.

The NMF package implements this procedure in methods `featureScore` and `extractFeature`:

```
# only compute the scores
s <- featureScore(res)
summary(s)

##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0001208 0.0162686 0.0548891 0.1185344 0.1209758 1.0000000

# compute the scores and characterize each metagene
s <- extractFeatures(res)
str(s)

## List of 3
##  $ : int [1:8] 39 74 2 91 167 190 103 174
##  $ : int [1:13] 94 1 112 42 8 64 96 182 59 41 ...
##  $ : int [1:5] 43 120 128 130 129
##  - attr(*, "method")= chr "kim"
```

2.2 Specifying the algorithm

2.2.1 Built-in algorithms

The *NMF* package package provides a number of built-in algorithms, that are listed or retrieved by function `nmfAlgorithm`. Each algorithm is identified by a unique name. The following algorithms are currently implemented (cf. Table 1 for more details):

```
nmfAlgorithm()

##  [1] "brunet"    "KL"        "lee"       "Frobenius" "offset"    "nsNMF"
##  [7] "ls-nmf"    "pe-nmf"    "siNMF"     "snmf/r"    "snmf/l"
```

The algorithm used to compute the NMF is specified in the third argument (`method`). For example, to use the NMF algorithm from Lee and Seung (Lee et al. 2001) based on the Frobenius euclidean norm, one make the following call:

```
# using Lee and Seung's algorithm
res <- nmf(esGolub, 3, 'lee')
algorithm(res)

## [1] "lee"
```

To use the Nonsmooth NMF algorithm from (Pascual-Montano et al. 2006):

```
# using the Nonsmooth NMF algorithm with parameter theta=0.7
res <- nmf(esGolub, 3, 'ns', theta=0.7)
algorithm(res)

## [1] "nsNMF"

fit(res)

## <Object of class:NMFns>
## features: 200
## basis/rank: 3
## samples: 38
## theta: 0.7
```

Or to use the PE-NMF algorithm from (Zhang et al. 2008):

```
# using the PE-NMF algorithm with parameters alpha=0.01, beta=1
res <- nmf(esGolub, 3, 'pe', alpha=0.01, beta=1)
res

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 200
##   basis/rank: 3
##   samples: 38
##  # Details:
##   algorithm:  pe-nmf
##   seed:  random
##   RNG: 10403L, 270L, ..., -1891789767L [9c04e7ae51353a97aaa7ee4a9351c264]
##   distance metric:  <function>
##   residuals:  67.35798
##   parameters: alpha=0.01, beta=1
##   Iterations: 2000
##   Timing:
##      user  system elapsed
##     0.663   0.003   0.666
```

2.2.2 Custom algorithms

The *NMF* package package provides the user the possibility to define his own algorithms, and benefit from all the functionalities available in the NMF framework. There are only few contraints on the way the custom algorithm must be defined. See the details in Section 3.1.1.

2.3 Specifying the seeding method

The seeding method used to compute the starting point for the chosen algorithm can be set via argument `seed`. Note that if the seeding method is deterministic there is no need to perform multiple run anymore.

2.3.1 Built-in seeding methods

Similarly to the algorithms, the `nmfSeed` function can be used to list or retrieve the built-in seeding methods. The following seeding methods are currently implemented:

```
nmfSeed()

## [1] "none"   "random" "ica"    "nndsvd"
```

To use a specific method to seed the computation of a factorization, one simply passes its name to `nmf`:

```
res <- nmf(esGolub, 3, seed='nndsvd')
res

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 200
##   basis/rank: 3
##   samples: 38
##  # Details:
##   algorithm:  brunet
##   seed:  nndsvd
##   RNG: 10403L, 360L, ..., -1662044055L [e2af28ae027e4131346cdde9fd9cd9b8]
##   distance metric:  'KL'
##   residuals:  547143.5
##   Iterations: 1090
##   Timing:
##      user  system elapsed
##     0.351   0.000   0.351
```

2.3.2 Numerical seed

Another possibility, useful when comparing methods or reproducing results, is to set the random number generator (RNG) by passing a numerical value in argument `seed`. This value is used to set the state of the RNG, and the initialization is performed by the built-in seeding method `'random'`. When the function `nmf` exits, the value of the random seed (`.Random.seed`) is restored to its original state – as before the call.

In the case of a single run (i.e. with `nrun=1`), the default is to use the current RNG, set with the R core function `set.seed`. In the case of multiple runs, the computations use RNGstream, as provided by the core RNG "L'Ecuyer-CMRG" (L'Ecuyer et al. 2002), which generates multiple independent random streams (one per run). This ensures the complete reproducibility of any given

set of runs, even when their computation is performed in parallel. Since RNGstream requires a 6-length numeric seed, a random one is generated if only a single numeric value is passed to `seed`. Moreover, single runs can also use RNGstream by passing a 6-length seed.

```
# single run and single numeric seed
res <- nmf(esGolub, 3, seed=123456)
showRNG(res)

## # RNG kind:  Mersenne-Twister / Inversion / Rejection
## # RNG state: 10403L, 624L, ..., 449848215L [825a7345133c6f7869527d185527e323]

# multiple runs and single numeric seed
res <- nmf(esGolub, 3, seed=123456, nrun=2)
showRNG(res)

## # RNG kind:  L'Ecuyer-CMRG / Inversion / Rejection
## # RNG state: 10407L, -1896287838L, -649582056L, -453180354L, -16866084L, -2023648666L, -827768

# single run with a 6-length seed
res <- nmf(esGolub, 3, seed=rep(123456, 6))
showRNG(res)

## # RNG kind:  L'Ecuyer-CMRG / Inversion / Rejection
## # RNG state: 10407L, 123456L, 123456L, 123456L, 123456L, 123456L, 123456L
```

NB: To show the RNG changes happening during the computation use `.options='v4'` to turn on verbosity at level 4.
In versions prior 0.6, one could specify option `restore.seed=FALSE` or `'-r'`, this option is now deprecated.

2.3.3 Fixed factorization

Yet another option is to completely specify the initial factorization, by passing values for matrices W and H:

```
# initialize a "constant" factorization based on the target dimension
init <- nmfModel(3, esGolub, W=0.5, H=0.3)
head(basis(init))

##                [,1] [,2] [,3]
## M12759_at       0.5  0.5  0.5
## U46006_s_at     0.5  0.5  0.5
## X70083_at       0.5  0.5  0.5
## X03100_cds2_at  0.5  0.5  0.5
## L32976_at       0.5  0.5  0.5
## M19878_s_at     0.5  0.5  0.5

# fit using this NMF model as a seed
res <- nmf(esGolub, 3, seed=init)
```

2.3.4 Custom function

The *NMF* package package provides the user the possibility to define his own seeding method, and benefit from all the functionalities available in the NMF framework. There are only few contraints on the way the custom seeding method must be defined. See the details in Section 3.2.

2.4 Multiple runs

When the seeding method is stochastic, multiple runs are usually required to achieve stability or a resonable result. This can be done by setting argument `nrun` to the desired value. For performance reason we use `nrun=5` here, but a typical choice would lies between 100 and 200:

```
res.multirun <- nmf(esGolub, 3, nrun=5)
res.multirun

## <Object of class: NMFfitX1 >
##   Method: brunet
##   Runs:  5
##   RNG:
##    10407L, 2060366693L, -1979266846L, -1467721989L, 23029248L, 1469230593L, 1095525326L
##   Total timing:
##    user  system elapsed
##   2.348   0.278   2.870
```

By default, the returned object only contains the best fit over all the runs. That is the factorization that achieved the lowest approximation error (i.e. the lowest objective value). Even during the computation, only the current best factorization is kept in memory. This limits the memory requirement for performing multiple runs, which in turn allows to perform more runs.

The object `res.multirun` is of class `NMFfitX1` that extends class `NMFfit`, the class returned by single NMF runs. It can therefore be handled as the result of a single run and benefit from all the methods defined for single run results.

If one is interested in keeping the results from all the runs, one can set the option `keep.all=TRUE`:

```
# explicitly setting the option keep.all to TRUE
res <- nmf(esGolub, 3, nrun=5, .options=list(keep.all=TRUE))
res

## <Object of class: NMFfitXn >
##   Method: brunet
##   Runs:  5
##   RNG:
##    10407L, 1997333926L, -492612625L, 469522788L, -1828607723L, 1996453586L, -1167273941L
##   Total timing:
##    user  system elapsed
##   1.687   0.246   2.311
##   Sequential timing:
##    user  system elapsed
##   0.879   0.002   0.880
```

```
# or using letter code 'k' in argument .options
nmf(esGolub, 3, nrun=5, .options='k')
```

In this case, the result is an object of class `NMFfitXn` that also inherits from class `list`.

Note that keeping all the results may be memory consuming. For example, a 3-rank `NMF` fit[8] for the Golub gene expression matrix (5000×38) takes about 28Kb[9].

[8] i.e. the result of a single NMF run with rank equal 3.

[9] This size might change depending on the architecture (32 or 64 bits)

2.5 Parallel computations

To speed-up the analysis whenever possible, the *NMF* package package implements transparent parallel computations when run on multi-core machines. It uses the `foreach` framework developed by REvolution Computing *foreach* package[10] (Microsoft et al. 2020), together with the related `doParallel` parallel backend from the *doParallel* package[11] (Corporation et al. 2019) – based on the *parallel* package – to make use of all the CPUs available on the system, with each core simultaneously performing part of the runs.

2.5.1 Memory considerations

Running multicore computations increases the required memory linearly with the number of cores used. When only the best run is of interest, memory usage is optimized to only keep the current best factorization. On non-Windows machine, further speed improvement are achieved by using shared memory and mutex objects from the *bigmemory* package[12] (Kane et al. 2013a) and the *synchronicity* package[13] (Kane et al. 2013b).

2.5.2 Parallel foreach backends

The default parallel backend used by the `nmf` function is defined by the package specific option `'pbackend'`, which defaults to `'par'` – for `doParallel`. The backend can also be set on runtime via argument `.pbackend`.

IMPORTANT NOTE: The parallel computation is based on the *doParallel* and *parallel* packages, and the same care should be taken as stated in the vignette of the *doMC*:

> *... it usually isn't safe to run doMC and multicore from a GUI environment. In particular, it is not safe to use doMC from R.app on Mac OS X. Instead, you should use doMC from a terminal session, starting R from the command line.*

Therefore, the `nmf` function does not allow to run multicore computation from the MacOS X GUI.

From version 0.8, other parallel backends are supported, and may be specified via argument `.pbackend`:

`.pbackend='mpi'` uses the parallel backend *doParallel* package[14] (Corporation et al. 2019) and *doMPI* package[15] (Weston 2017)

`.pbackend=NULL`

It is possible to specify that the currently registered backend should be used, by setting argument `.pbackend=NULL`. This allow to perform parallel computations with "permanent" backends that are configured externally of the `nmf` call.

2.5.3 Runtime options

There are two other runtime options, `parallel` and `parallel.required`, that can be passed via argument `.options`, to control the behaviour of the parallel computation (see below).

A call for multiple runs will be computed in parallel if one of the following condition is satisfied:

[10] https://cran.r-project.org/package=foreach
[11] https://cran.r-project.org/package=doParallel
[12] https://cran.r-project.org/package=bigmemory
[13] https://cran.r-project.org/package=synchronicity
[14] https://cran.r-project.org/package=doParallel
[15] https://cran.r-project.org/package=doMPI

- call with option `'P'` or `parallel.required` set to TRUE (note the upper case in `'P'`). In this case, if for any reason the computation cannot be run in parallel (packages requirements, OS, ...), then an error is thrown. Use this mode to force the parallel execution.
- call with option `'p'` or `parallel` set to TRUE. In this case if something prevents a parallel computation, the factorizations will be done sequentially.
- a valid parallel backend is specified in argument `.pbackend`. For the moment it can either be the string `'mc'` or a single `numeric` value specifying the number of core to use. Unless option `'P'` is specified, it will run using option `'p'` (i.e. try-parallel mode).

NB: The number of processors to use can also be specified in the runtime options as e.g. `.options='p4'` or `.options='P4'` – to ask or request 4 CPUs.

Examples
The following exmaples are run with `.options='v'` which turn on verbosity at level 1, that will show which parallell setting is used by each computation. Although we do not show the output here, the user is recommended to run these commands on his machine to see the internal differences of each call.

```
# the default call will try to run in parallel using all the cores
# => will be in parallel if all the requirements are satisfied
nmf(esGolub, 3, nrun=5, .opt='v')

# request a certain number of cores to use => no error if not possible
nmf(esGolub, 3, nrun=5, .opt='vp8')

# force parallel computation: use option 'P'
nmf(esGolub, 3, nrun=5, .opt='vP')

# require an improbable number of cores => error
nmf(esGolub, 3, nrun=5, .opt='vP200')
```

2.5.4 High Performance Computing on a cluster

To achieve further speed-up, the computation can be run on an HPC cluster. In our tests we used the *doMPI* package[16] (Weston 2017) to perform 100 factorizations using hybrid parallel computation on 4 quadri-core machines – making use of all the cores computation on each machine.

```
# file: mpi.R

## 0. Create and register an MPI cluster
library(doMPI)
cl <- startMPIcluster()
registerDoMPI(cl)
library(NMF)

# run on all workers using the current parallel backend
data(esGolub)
res <- nmf(esGolub, 3, 'brunet', nrun=n, .opt='p', .pbackend=NULL)
```

[16] https://cran.r-project.org/package=doMPI

```
# save result
save(res, file='result.RData')

## 4. Shutdown the cluster and quit MPI
closeCluster(cl)
mpi.quit()
```

Passing the following shell script to *qsub* should launch the execution on a Sun Grid Engine HPC cluster, with OpenMPI. Some adaptation might be necessary for other queueing systems/installations.

```
#!/bin/bash
#$ -cwd
#$ -q opteron.q
#$ -pe mpich_4cpu 16
echo "Got $NSLOTS slots. $TMP/machines"

orterun -v -n $NSLOTS -hostfile $TMP/machines R --slave -f mpi.R
```

2.5.5 Forcing sequential execution

When running on a single core machine, *NMF* package package has no other option than performing the multiple runs sequentially, one after another. This is done via the `sapply` function.

On multi-core machine, one usually wants to perform the runs in parallel, as it speeds up the computation (cf. Section 2.5). However in some situation (e.g. while debugging), it might be useful to force the sequential execution of the runs. This can be done via the option `'p1'` to run on a single core , or with `.pbackend='seq'` to use the foreach backend `doSEQ` or to `NA` to use a standard `sapply` call:

```
# parallel execution on 2 cores (if possible)
res1 <- nmf(esGolub, 3, nrun=5, .opt='vp2', seed=123)

## NMF algorithm:  'brunet'
## Multiple runs:  5
## Mode:  parallel (2/8 core(s))

##
Runs: |
Runs: |                                                   |   0%
Runs: |
Runs: |==================================================| 100%
## System time:
##    user  system elapsed
##   2.179   0.210   2.629

# or use the doParallel with single core
res2 <- nmf(esGolub, 3, nrun=5, .opt='vp1', seed=123)

## NMF algorithm:  'brunet'
## Multiple runs:  5
## Mode:  sequential [foreach:doParallelMC]
```

```
##
Runs: |
Runs: |                                                          |   0%
Runs: |
Runs: |========                                                  |  17%
Runs: |
Runs: |================                                          |  33%
Runs: |
Runs: |========================                                  |  50%
Runs: |
Runs: |================================                          |  67%
Runs: |
Runs: |========================================                  |  83%
Runs: |
Runs: |================================================          | 100%
## System time:
##    user  system elapsed
##   2.922   0.000   2.923

# force sequential computation by sapply: use option '-p' or .pbackend=NA
res3 <- nmf(esGolub, 3, nrun=5, .opt='v-p', seed=123)

## NMF algorithm:  'brunet'
## Multiple runs:  5
## Mode:  sequential [sapply]

## Runs: 1* 2* 3* 4 5 ... DONE
## System time:
##    user  system elapsed
##   1.497   0.000   1.497

res4 <- nmf(esGolub, 3, nrun=5, .opt='v', .pbackend=NA, seed=123)

## NMF algorithm:  'brunet'
## Multiple runs:  5
## Mode:  sequential [sapply]

## Runs: 1* 2* 3* 4 5 ... DONE
## System time:
##    user  system elapsed
##   1.490   0.000   1.489

# or use the SEQ backend of foreach: .pbackend='seq'
res5 <- nmf(esGolub, 3, nrun=5, .opt='v', .pbackend='seq', seed=123)

## NMF algorithm:  'brunet'
## Multiple runs:  5
## Mode:  sequential [foreach:doSEQ]

##
Runs: |
Runs: |                                                          |   0%
Runs: |
Runs: |========                                                  |  17%
Runs: |
Runs: |================                                          |  33%
```

```
Runs: |
Runs: |=========================                              |  50%
Runs: |
Runs: |==================================                     |  67%
Runs: |
Runs: |==========================================             |  83%
Runs: |
Runs: |================================================== 100%
## System time:
##    user  system elapsed
##   2.863   0.000   2.864

# all results are all identical
nmf.equal(list(res1, res2, res3, res4, res5))

## [1] TRUE
```

2.6 Estimating the factorization rank

A critical parameter in NMF is the factorization rank r. It defines the number of metagenes used to approximate the target matrix. Given a NMF method and the target matrix, a common way of deciding on r is to try different values, compute some quality measure of the results, and choose the best value according to this quality criteria.

Several approaches have then been proposed to choose the optimal value of r. For example, (Brunet et al. 2004) proposed to take the first value of r for which the cophenetic coefficient starts decreasing, (Hutchins et al. 2008) suggested to choose the first value where the RSS curve presents an inflection point, and (Frigyesi et al. 2008) considered the smallest value at which the decrease in the RSS is lower than the decrease of the RSS obtained from random data.

The *NMF* package package provides functions to help implement such procedures and plot the relevant quality measures. Note that this can be a lengthy computation, depending on the data size. Whereas the standard NMF procedure usually involves several hundreds of random initialization, performing 30-50 runs is considered sufficient to get a robust estimate of the factorization rank (Brunet et al. 2004; Hutchins et al. 2008). For performance reason, we perform here only 10 runs for each value of the rank.

```
# perform 10 runs for each value of r in range 2:6
estim.r <- nmf(esGolub, 2:6, nrun=10, seed=123456)
```

The result is a S3 object of class `NMF.rank`, that contains a `data.frame` with the quality measures in column, and the values of r in row. It also contains a list of the consensus matrix for each value of r.

All the measures can be plotted at once with the method `plot` (Figure 1), and the function `consensusmap` generates heatmaps of the consensus matrix for each value of the rank. In the context of class discovery, it is useful to see if the clusters obtained correspond to known classes. This is why in the particular case of the Golub dataset, we added annotation tracks for the two covariates available ('Cell' and 'ALL.AML'). Since we removed the variable 'Sample' in the preliminaries, these are the only variables in the phenotypic `data.frame` embedded within the `ExpressionSet` object, and we can simply pass the whole object to argument `annCol` (Figure 2). One can see that at rank 2, the clusters correspond to the ALL and AML samples respectively, while rank 3 separates AML from ALL/T-cell and ALL/B-cell[17].

[17] Remember that the plots shown in Figure 2 come from only 10 runs, using the 200 first genes in the dataset, which explains the somewhat not so clean clusters. The results are in fact much cleaner when using the full dataset (Figure 6).

```
plot(estim.r)
```

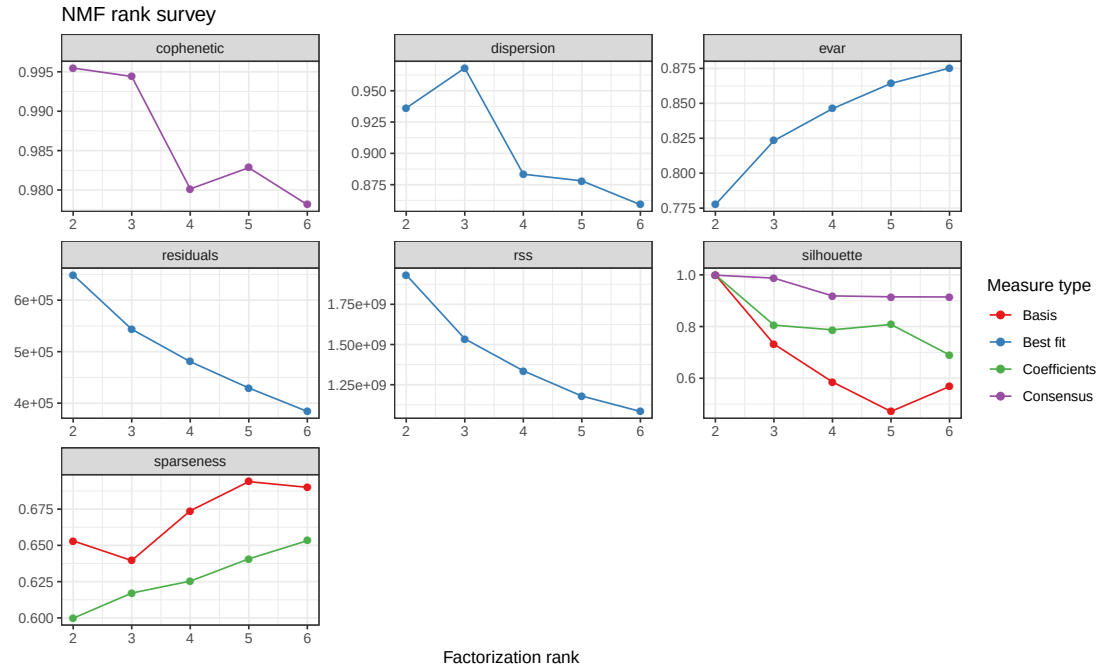

Figure 1: Estimation of the rank: Quality measures computed from 10 runs for each value of r.

```
consensusmap(estim.r, annCol=esGolub, labCol=NA, labRow=NA)
```

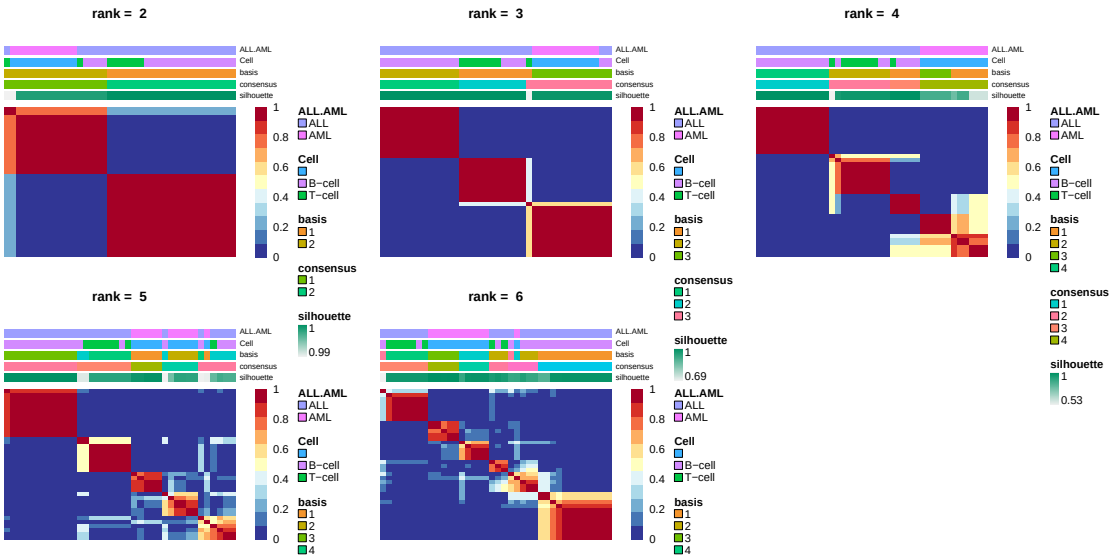

Figure 2: Estimation of the rank: Consensus matrices computed from 10 runs for each value of r.

```r
# shuffle original data
V.random <- randomize(esGolub)
# estimate quality measures from the shuffled data (use default NMF algorithm)
estim.r.random <- nmf(V.random, 2:6, nrun=10, seed=123456)
# plot measures on same graph
plot(estim.r, estim.r.random)
```

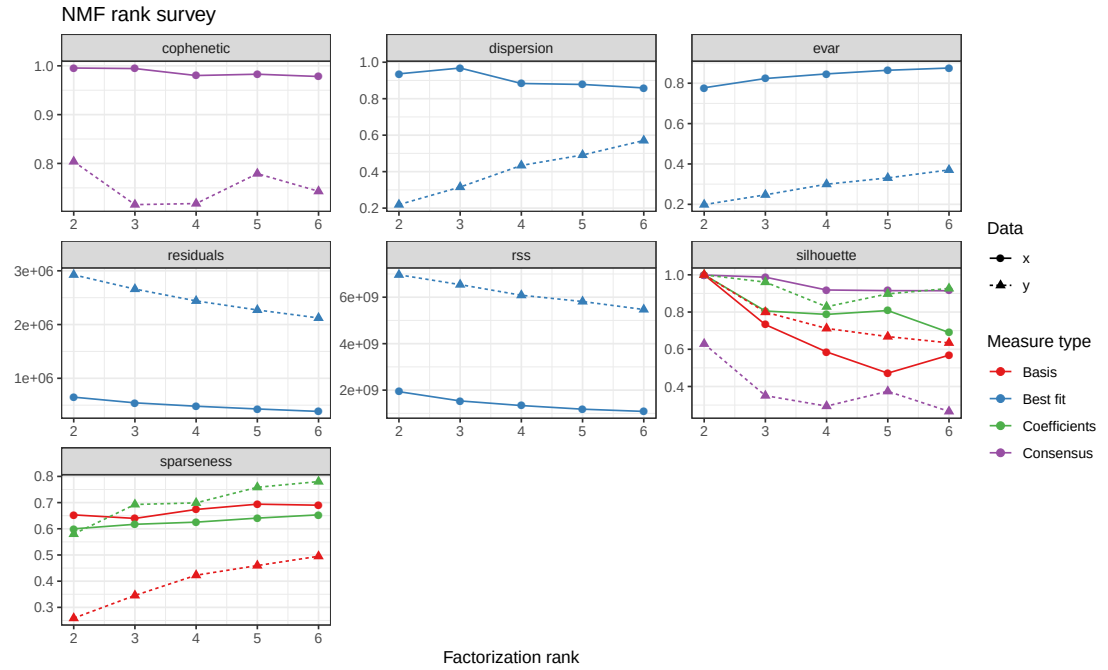

Figure 3: Estimation of the rank: Comparison of the quality measures with those obtained from randomized data. The curves for the actual data are in blue and green, those for the randomized data are in red and pink. The estimation is based on Brunet's algorithm.

2.6.1 Overfitting

Even on random data, increasing the factorization rank would lead to decreasing residuals, as more variables are available to better fit the data. In other words, there is potentially an overfitting problem.

In this context, the approach from (Frigyesi et al. 2008) may be useful to prevent or detect overfitting as it takes into account the results for unstructured data. However it requires to compute the quality measure(s) for the random data. The *NMF* package package provides a function that shuffles the original data, by permuting the rows of each column, using each time a different permutation. The rank estimation procedure can then be applied to the randomized data, and the "random" measures added to the plot for comparison (Figure 3).

2.7 Comparing algorithms

To compare the results from different algorithms, one can pass a list of methods in argument `method`. To enable a fair comparison, a deterministic seeding method should also be used. Here we fix the random seed to 123456.

```r
# fit a model for several different methods
res.multi.method <- nmf(esGolub, 3, list('brunet', 'lee', 'ns'), seed=123456, .options='t')
```

```
## Compute NMF method 'brunet' [1/3] ... OK
## Compute NMF method 'lee' [2/3] ... OK
## Compute NMF method 'ns' [3/3] ... OK
```

Passing the result to method `compare` produces a `data.frame` that contains summary measures for each method. Again, prior knowledge of classes may be used to compute clustering quality measures:

```
compare(res.multi.method)

##        method   seed rng    metric rank sparseness.basis sparseness.coef
## brunet brunet random   1        KL    3        0.6392676       0.6217884
## lee       lee random   1 euclidean    3        0.7268875       0.4465608
## nsNMF   nsNMF random   1        KL    3        0.6777185       0.7350386
##        silhouette.coef silhouette.basis   residuals niter   cpu cpu.all nrun
## brunet       0.8126484        0.7487792    543535.7   510 0.129   0.129    1
## lee          0.7147803        0.6764436 673513120.5  1780 0.471   0.471    1
## nsNMF        0.8654196        0.7946175    585106.4   970 0.391   0.391    1

# If prior knowledge of classes is available
compare(res.multi.method, class=esGolub$Cell)

##        method   seed rng    metric rank sparseness.basis sparseness.coef
## brunet brunet random   1        KL    3        0.6392676       0.6217884
## lee       lee random   1 euclidean    3        0.7268875       0.4465608
## nsNMF   nsNMF random   1        KL    3        0.6777185       0.7350386
##           purity   entropy silhouette.coef silhouette.basis   residuals niter
## brunet 0.8157895 0.3926954       0.8126484        0.7487792    543535.7   510
## lee    0.5789474 0.7231282       0.7147803        0.6764436 673513120.5  1780
## nsNMF  0.7894737 0.4691212       0.8654196        0.7946175    585106.4   970
##          cpu cpu.all nrun
## brunet 0.129   0.129    1
## lee    0.471   0.471    1
## nsNMF  0.391   0.391    1
```

Because the computation was performed with error tracking enabled, an error plot can be produced by method `plot` (Figure 4). Each track is normalized so that its first value equals one, and stops at the iteration where the method's convergence criterion was fulfilled.

2.8 Visualization methods

Error track

If the NMF computation is performed with error tracking enabled – using argument `.options` – the trajectory of the objective value is computed during the fit. This computation is not enabled by default as it induces some overhead.

```
# run nmf with .option='t'
res <- nmf(esGolub, 3, .options='t')
# or with .options=list(track=TRUE)
```

The trajectory can be plot with the method `plot` (Figure 4):

```
plot(res)
plot(res.multi.method)
```

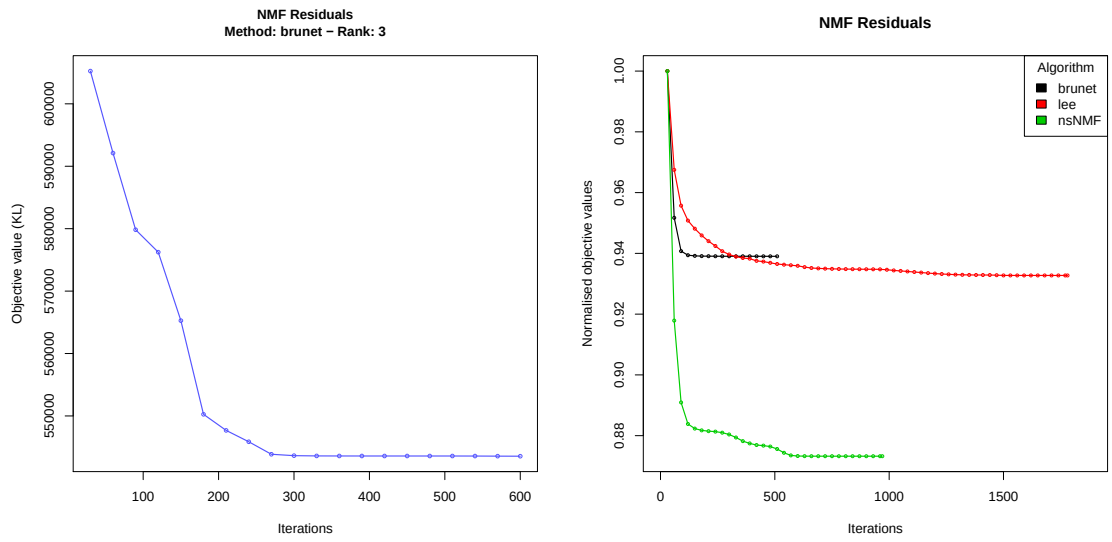

Figure 4: Error track for a single NMF run (left) and multiple method runs (right)

Heatmaps

The methods `basismap`, `coefmap` and `consensusmap` provide an easy way to visualize respectively the resulting basis matrix (i.e. metagenes), mixture coefficient matrix (i.e. metaprofiles) and the consensus matrix, in the case of multiple runs. It produces pre-configured heatmaps based on the function `aheatmap`, the underlying heatmap engine provided with the package NMF. The default heatmaps produced by these functions are shown in Figures 5 and 6. They can be customized in many different ways (colours, annotations, labels). See the dedicated vignette *"NMF: generating heatmaps"* or the help pages `?coefmap` and `?aheatmap` for more information.

An important and unique feature of the function `aheatmap`, is that it makes it possible to combine several heatmaps on the same plot, using the both standard layout calls `par(mfrow=...)` and `layout(...)`, or grid viewports from `grid` graphics. The plotting context is automatically internally detected, and a correct behaviour is achieved thanks to the *gridBase* package[18] (Murrell 2014). Examples are provided in the dedicated vignette mentioned above.

The rows of the basis matrix often carry the high dimensionality of the data: genes, loci, pixels, features, etc... The function `basismap` extends the use of argument `subsetRow` (from `aheatmap`) to the specification of a feature selection method. In Figure 5 we simply used `subsetRow=TRUE`, which subsets the rows using the method described in (Kim et al. 2007), to only keep the basis-specific features (e.g. the metagene-specific genes). We refer to the relevant help pages `?basismap` and `?aheatmap` for more details about other possible values for this argument.

In the case of multiple runs the function `consensusmap` plots the consensus matrix, i.e. the average connectivity matrix across the runs (see results in Figure 6 for a consensus matrix obtained with 100 runs of Brunet's algorithm on the complete Golub dataset):

3 Extending the package

We developed the *NMF* package package with the objective to facilitate the integration of new NMF methods, trying to impose only few requirements on their implementations. All the built-in

[18] https://cran.r-project.org/package=gridBase

```
layout(cbind(1,2))
# basis components
basismap(res, subsetRow=TRUE)
# mixture coefficients
coefmap(res)
```

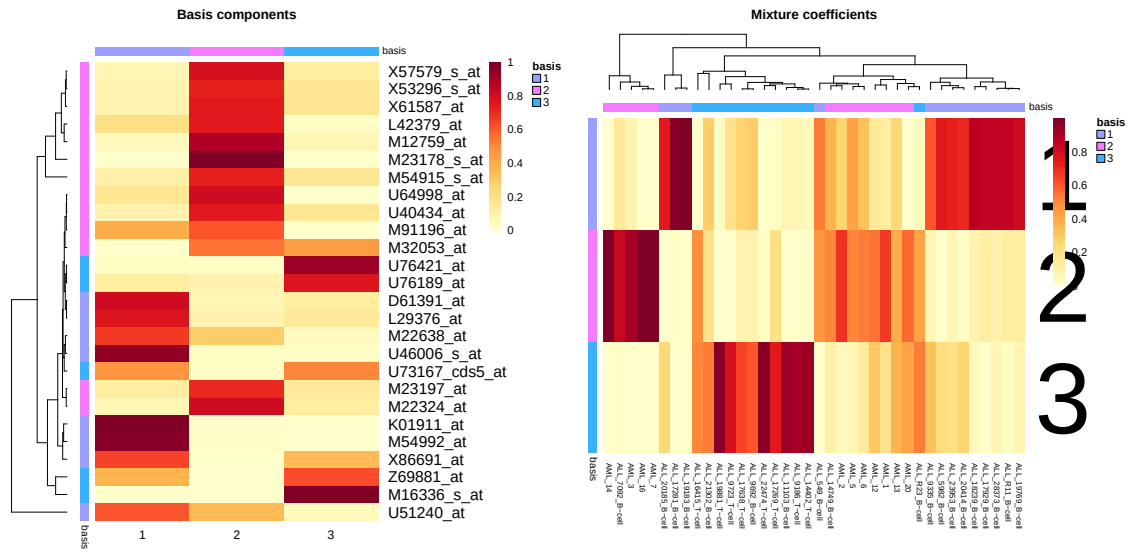

Figure 5: Heatmap of the basis and the mixture coefficient matrices. The rows of the basis matrix were selected using the default feature selection method – described in (Kim et al. 2007).

algorithms and seeding methods are implemented as strategies that are called from within the main interface method `nmf`.

The user can define new strategies and those are handled in exactly the same way as the built-in ones, benefiting from the same utility functions to interpret the results and assess their performance.

3.1 Custom algorithm

3.1.1 Using a custom algorithm

To define a strategy, the user needs to provide a `function` that implements the complete algotihm. It must be of the form:

```
my.algorithm <- function(x, seed, param.1, param.2){
        # do something with starting point
        # ...

        # return updated starting point
        return(seed)
}
```

Where:

target is a `matrix`;

start is an object that inherits from class `NMF`. This `S4` class is used to handle NMF models (matrices `W` and `H`, objective function, etc...);

param.1, param.2 are extra parameters specific to the algorithms;

The function must return an object that inherits from class `NMF`.
For example:

```
my.algorithm <- function(x, seed, scale.factor=1){
        # do something with starting point
        # ...
        # for example:
        # 1. compute principal components
        pca <- prcomp(t(x), retx=TRUE)

        # 2. use the absolute values of the first PCs for the metagenes
        # Note: the factorization rank is stored in object 'start'
        factorization.rank <- nbasis(seed)
        basis(seed) <- abs(pca$rotation[,1:factorization.rank])
        # use the rotated matrix to get the mixture coefficient
        # use a scaling factor (just to illustrate the use of extra parameters)
        coef(seed) <- t(abs(pca$x[,1:factorization.rank])) / scale.factor

        # return updated data
        return(seed)
}
```

To use the new method within the package framework, one pass `my.algorithm` to main interface `nmf` via argument `method`. Here we apply the algorithm to some matrix `V` randomly generated:

```
n <- 50; r <- 3; p <- 20
V <-syntheticNMF(n, r, p)
```

```
nmf(V, 3, my.algorithm, scale.factor=10)

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 50
##   basis/rank: 3
##   samples: 20
##  # Details:
##   algorithm:  nmf_6e3824ef8643
##   seed:  random
##   RNG: 10403L, 459L, ..., -419496537L [6057ec119f63baf6773461376d4e5e22]
##   distance metric:  'euclidean'
##   residuals:  492.4497
##   parameters: scale.factor=10
##   Timing:
##      user  system elapsed
##     0.004   0.000   0.004
```

3.1.2 Using a custom distance measure

The default distance measure is based on the euclidean distance. If the algorithm is based on another distance measure, this one can be specified in argument `objective`, either as a `character`

string corresponding to a built-in objective function, or a custom `function` definition[19]:

```
# based on Kullback-Leibler divergence
nmf(V, 3, my.algorithm, scale.factor=10, objective='KL')

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 50
##   basis/rank: 3
##   samples: 20
##  # Details:
##   algorithm:  nmf_6e3898e078e
##   seed:  random
##   RNG: 10403L, 45L, ..., 1096266081L [2bba3dd48c10aad2ebbb49eaa96efe62]
##   distance metric:  'KL'
##   residuals:  1491.885
##   parameters: scale.factor=10
##   Timing:
##      user  system elapsed
##     0.010   0.000   0.011

# based on custom distance metric
nmf(V, 3, my.algorithm, scale.factor=10
        , objective=function(model, target, ...){
            ( sum( (target-fitted(model))^4 ) )^{1/4}
                }
)

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 50
##   basis/rank: 3
##   samples: 20
##  # Details:
##   algorithm:  nmf_6e385dc1011e
##   seed:  random
##   RNG: 10403L, 255L, ..., 1096266081L [0447c2aead14d877b7393c7fa4ca996c]
##   distance metric:  <function>
##   residuals:  8.41365
##   parameters: scale.factor=10
##   Timing:
##      user  system elapsed
##     0.002   0.000   0.002
```

3.1.3 Defining algorithms for mixed sign data

All the algorithms implemented in the *NMF* package package assume that the input data is nonnegative. However, some methods exist in the litterature that work with relaxed constraints, where the input data and one of the matrix factors (W or H) are allowed to have negative entries (eg. semi-NMF (Ding et al. 2010; Roux et al. 2008)). Strictly speaking these methods do not

[19]Note that from version 0.8, the arguments for custom objective functions have been swapped: (1) the current NMF model, (2) the target matrix

fall into the NMF category, but still solve constrained matrix factorization problems, and could be considered as NMF methods when applied to non-negative data. Moreover, we received user requests to enable the development of semi-NMF type methods within the package's framework. Therefore, we designed the *NMF* package package so that such algorithms – that handle negative data – can be integrated. This section documents how to do it.

By default, as a safe-guard, the sign of the input data is checked before running any method, so that the `nmf` function throws an error if applied to data that contain negative entries [20]. To extend the capabilities of the *NMF* package package in handling negative data, and plug mixed sign NMF methods into the framework, the user needs to specify the argument `mixed=TRUE` in the call to the `nmf` function. This will skip the sign check of the input data and let the custom algorithm perform the factorization.

As an example, we reuse the previously defined custom algorithm[21]:

```
# put some negative input data
V.neg <- V; V.neg[1,] <- -1;

# this generates an error
try( nmf(V.neg, 3, my.algorithm, scale.factor=10) )

## Error:  NMF::nmf - Input matrix x contains some negative entries.

# this runs my.algorithm without error
nmf(V.neg, 3, my.algorithm, mixed=TRUE, scale.factor=10)

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 50
##   basis/rank: 3
##   samples: 20
##  # Details:
##   algorithm:  nmf_6e385189de2c
##   seed:  random
##   RNG: 10403L, 465L, ..., 1096266081L [ea8ff0cac361311538ba61100c10709c]
##   distance metric:  'euclidean'
##   residuals:  489.997
##   parameters: scale.factor=10
##   Timing:
##      user  system elapsed
##     0.002   0.000   0.002
```

3.1.4 Specifying the NMF model

If not specified in the call, the NMF model that is used is the standard one, as defined in Equation (1). However, some NMF algorithms have different underlying models, such as non-smooth NMF (Pascual-Montano et al. 2006) which uses an extra matrix factor that introduces an extra parameter, and change the way the target matrix is approximated.

The NMF models are defined as S4 classes that extends class `NMF`. All the available models can be retreived calling the `nmfModel()` function with no argument:

[20] Note that on the other side, the sign of the factors returned by the algorithms is never checked, so that one can always return factors with negative entries.

[21] As it is defined here, the custom algorithm still returns nonnegative factors, which would not be desirable in a real example, as one would not be able to closely fit the negative entries.

```
nmfModel()

## <Object of class:NMFstd>
## features: 0
## basis/rank: 0
## samples: 0
```

One can specify the NMF model to use with a custom algorithm, using argument `model`. Here we first adapt a bit the custom algorithm, to justify and illustrate the use of a different model. We use model `NMFOffset` (Badea 2008), that includes an offset to take into account genes that have constant expression levels accross the samples:

```
my.algorithm.offset <- function(x, seed, scale.factor=1){
        # do something with starting point
        # ...
        # for example:
        # 1. compute principal components
        pca <- prcomp(t(x), retx=TRUE)

        # retrieve the model being estimated
        data.model <- fit(seed)

        # 2. use the absolute values of the first PCs for the metagenes
        # Note: the factorization rank is stored in object 'start'
        factorization.rank <- nbasis(data.model)
        basis(data.model) <- abs(pca$rotation[,1:factorization.rank])
        # use the rotated matrix to get the mixture coefficient
        # use a scaling factor (just to illustrate the use of extra parameters)
        coef(data.model) <- t(abs(pca$x[,1:factorization.rank])) / scale.factor

        # 3. Compute the offset as the mean expression
        data.model@offset <- rowMeans(x)

        # return updated data
        fit(seed) <- data.model
        seed
}
```

Then run the algorithm specifying it needs model `NMFOffset`:

```
# run custom algorithm with NMF model with offset
nmf(V, 3, my.algorithm.offset, model='NMFOffset', scale.factor=10)

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFOffset>
##   features: 50
##   basis/rank: 3
##   samples: 20
##   offset: [ 0.8454748 0.8168729 0.4680196 0.2121849 0.6002035 ... ]
##  # Details:
##   algorithm:  nmf_6e3845ce9ffe
##   seed:  random
##   RNG: 10403L, 51L, ..., 959036133L [e43715ae36fc01464a888994f241d3bc]
```

```
##   distance metric:  'euclidean'
##   residuals:  300.2108
##   parameters: scale.factor=10
##   Timing:
##      user  system elapsed
##     0.012   0.000   0.012
```

3.2 Custom seeding method

The user can also define custom seeding method as a function of the form:

```
# start: object of class NMF
# target: the target matrix
my.seeding.method <- function(model, target){

        # use only the largest columns for W
        w.cols <- apply(target, 2, function(x) sqrt(sum(x^2)))
        basis(model) <- target[,order(w.cols)[1:nbasis(model)]]

        # initialize H randomly
        coef(model) <- matrix(runif(nbasis(model)*ncol(target))
                                              , nbasis(model), ncol(target))

        # return updated object
        return(model)
}
```

To use the new seeding method:

```
nmf(V, 3, 'snmf/r', seed=my.seeding.method)

## <Object of class: NMFfit>
##  # Model:
##   <Object of class:NMFstd>
##   features: 50
##   basis/rank: 3
##   samples: 20
##  # Details:
##   algorithm:  snmf/r
##   seed:  NMF.seed.6e386881b607
##   RNG: 10403L, 311L, ..., 959036133L [160093b7fd363e6b40c878c355b4cdae]
##   distance metric:  <function>
##   residuals:  196.7261
##   Iterations: 65
##   Timing:
##      user  system elapsed
##     0.731   0.492   0.195
```

4 Advanced usage

4.1 Package specific options

The package specific options can be retieved or changed using the `nmf.getOption` and `nmf.options` functions. These behave similarly as the `getOption` and `nmf.options` base functions:

```
#show default algorithm and seeding method
nmf.options('default.algorithm', 'default.seed')

## $default.algorithm
## [1] "brunet"
##
## $default.seed
## [1] "random"

# retrieve a single option
nmf.getOption('default.seed')

## [1] "random"

## # All options
## nmf.options()
```

Currently the following options are available:

gc Interval/frequency (in number of runs) at which garbage collection is performed.

verbose Default level of verbosity.

debug Toogles debug mode. In this mode the console output may be very – very – messy, and is aimed at debugging only.

% end description

The default/current values of each options can be displayed using the function `nmf.printOptions`:

```
nmf.printOptions()

## <Package specific options: package:NMF>
## Registered: FALSE
## Options:
## List of 12
##  $ default.algorithm: chr "brunet"
##  $ default.seed     : chr "random"
##  $ error.track      : 'option_symlink' chr "track"
##  $ track            : logi FALSE
##  $ track.interval   : num 30
##  $ gc               : num 50
##  $ parallel.backend : 'option_symlink' chr "pbackend"
##  $ pbackend         : chr "par"
##  $ verbose          : logi FALSE
##  $ debug            : logi FALSE
##  $ shared.memory    : logi TRUE
##  $ cores            : num 2
## Defaults:
## List of 11
```

```
##  $ default.algorithm: chr "brunet"
##  $ default.seed     : chr "random"
##  $ error.track      : 'option_symlink' chr "track"
##  $ track            : logi FALSE
##  $ track.interval   : num 30
##  $ gc               : num 50
##  $ parallel.backend : 'option_symlink' chr "pbackend"
##  $ pbackend         : chr "par"
##  $ verbose          : logi FALSE
##  $ debug            : logi FALSE
##  $ shared.memory    : logi TRUE
```

5 Session Info

- R version 3.6.3 (2020-02-29), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_IL`, `LC_NUMERIC=C`, `LC_TIME=en_IL`, `LC_COLLATE=C`, `LC_MONETARY=en_IL`, `LC_MESSAGES=en_IL`, `LC_PAPER=en_IL`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_IL`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 19.10`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/openblas/libblas.so.3`
- LAPACK: `/usr/lib/x86_64-linux-gnu/libopenblasp-r0.3.7.so`
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, utils
- Other packages: Biobase 2.46.0, BiocGenerics 0.32.0, NMF 0.23.6, RColorBrewer 1.1-2, bigmemory 4.5.36, cluster 2.1.0, doParallel 1.0.15, foreach 1.4.8, iterators 1.0.12, knitr 1.28, pkgmaker 0.31.1, registry 0.5-1, rngtools 1.5.1, synchronicity 1.3.5, xtable 1.8-4
- Loaded via a namespace (and not attached): Matrix 1.2-18, R6 2.4.1, Rcpp 1.0.4, assertthat 0.2.1, bibtex 0.4.2.2, bigmemory.sri 0.1.3, codetools 0.2-16, colorspace 1.4-1, compiler 3.6.3, crayon 1.3.4, dendextend 1.13.4, digest 0.6.25, dplyr 0.8.5, evaluate 0.14, farver 2.0.3, ggplot2 3.3.0, glue 1.3.2, grid 3.6.3, gridBase 0.4-7, gridExtra 2.3, gtable 0.3.0, highr 0.8, labeling 0.3, lattice 0.20-40, lifecycle 0.2.0, magrittr 1.5, matrixStats 0.56.0, munsell 0.5.0, pillar 1.4.3, pkgconfig 2.0.3, plyr 1.8.6, purrr 0.3.3, reshape2 1.4.3, rlang 0.4.5, scales 1.1.0, stringi 1.4.6, stringr 1.4.0, tibble 2.1.3, tidyselect 1.0.0, tools 3.6.3, uuid 0.1-4, viridis 0.5.1, viridisLite 0.3.0, withr 2.1.2, xfun 0.12

References

[1] Russell Albright, James Cox, David Duling, Amy N. Langville, and C. Meyer. *Algorithms, initializations, and convergence for the nonnegative matrix factorization.* Tech. rep. 919. NCSU Technical Report Math 81706. http://meyer. math. ncsu. edu/Meyer/Abstracts/Publications. html, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.2161\&rep=rep1\&type=pdfhttp://meyer.math.ncsu.edu/Meyer/PS_Files/NMFInitAlgConv.pdf (see p. 5).

[2] Liviu Badea. "Extracting gene expression profiles common to colon and pancreatic adenocarcinoma using simultaneous nonnegative matrix factorization." In: *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing* 290 (2008), pp. 267–78. ISSN: 1793-5091. URL: http://www.ncbi.nlm.nih.gov/pubmed/18229692 (see pp. 5, 7, 29).

[3] M.W. Berry, M Browne, Amy N. Langville, V.P. Pauca, and R.J. Plemmons. "Algorithms and applications for approximate nonnegative matrix factorization". In: *Computational Statistics & Data Analysis* 52.1 (2007), pp. 155–173. URL: http://www.sciencedirect.com/science/article/pii/S0167947306004191 (see p. 4).

[4] C Boutsidis and E Gallopoulos. "SVD based initialization: A head start for nonnegative matrix factorization". In: *Pattern Recognition* 41.4 (2008), pp. 1350–1362. ISSN: 00313203. DOI: 10.1016/j.patcog.2007.09.010. URL: http://linkinghub.elsevier.com/retrieve/pii/S0031320307004359 (see p. 6).

[5] Jean-Philippe Brunet, Pablo Tamayo, Todd R Golub, and Jill P Mesirov. "Metagenes and molecular pattern discovery using matrix factorization." In: *Proceedings of the National Academy of Sciences of the United States of America* 101.12 (2004), pp. 4164–9. ISSN: 0027-8424. DOI: 10.1073/pnas.0308531101. URL: http://www.ncbi.nlm.nih.gov/pubmed/15016911 (see pp. 5, 6, 8, 20).

[6] M Chu, F Diele, R Plemmons, and S Ragni. "Optimality, computation, and interpretation of nonnegative matrix factorizations". In: *SIAM Journal on Matrix Analysis* (2004), pp. 4–8030. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.5758 (see p. 4).

[7] Andrzej Cichocki, Rafal Zdunek, and Shun ichi Amari. "Nonnegative matrix and tensor factorization". In: *IEEE Signal Processing Magazine* 25 (2008), pp. 142–145 (see p. 3).

[8] Microsoft Corporation and Steve Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.15. 2019. URL: https://CRAN.R-project.org/package=doParallel (see p. 16).

[9] Chris Ding, Tao Li, and Michael I Jordan. "Convex and semi-nonnegative matrix factorizations." In: *IEEE transactions on pattern analysis and machine intelligence* 32.1 (2010), pp. 45–55. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2008.277. URL: http://www.ncbi.nlm.nih.gov/pubmed/19926898 (see p. 27).

[10] Attila Frigyesi and Mattias Höglund. "Non-negative matrix factorization for the analysis of complex gene expression data: identification of clinically relevant tumor subtypes." In: *Cancer informatics* 6.2003 (2008), pp. 275–92. ISSN: 1176-9351. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2623306/ (see pp. 20, 22).

[11] Yuan Gao and George Church. "Improving molecular cancer class discovery through sparse non-negative matrix factorization." In: *Bioinformatics (Oxford, England)* 21.21 (2005), pp. 3970–5. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bti653. URL: http://www.ncbi.nlm.nih.gov/pubmed/16244221 (see p. 8).

[12] Renaud Gaujoux and Cathal Seoighe. "A flexible R package for nonnegative matrix factorization". In: *BMC Bioinformatics* 11.1 (2010), p. 367. ISSN: 1471-2105. DOI: 10.1186/1471-2105-11-367. URL: http://www.biomedcentral.com/1471-2105/11/367 (see p. 1).

[13] Robert C Gentleman et al. "Bioconductor: open software development for computational biology and bioinformatics." In: *Genome biology* 5.10 (2004), R80. ISSN: 1465-6914. DOI: 10.1186/gb-2004-5-10-r80. URL: http://www.ncbi.nlm.nih.gov/pubmed/15461798 (see p. 3).

[14] W. Huber et al. "Orchestrating high-throughput genomic analysis with Bioconductor". In: *Nature Methods* 12.2 (2015), pp. 115–121. URL: http://www.nature.com/nmeth/journal/v12/n2/full/nmeth.3252.html (see p. 6).

[15] Lucie N Hutchins, Sean M Murphy, Priyam Singh, and Joel H Graber. "Position-dependent motif characterization using non-negative matrix factorization." In: *Bioinformatics (Oxford, England)* 24.23 (2008), pp. 2684–90. ISSN: 1367-4811. DOI: 10.1093/bioinformatics/btn526. URL: http://www.ncbi.nlm.nih.gov/pubmed/18852176 (see p. 20).

[16] Michael J. Kane, John Emerson, and Stephen Weston. "Scalable Strategies for Computing with Massive Data". In: *Journal of Statistical Software* 55.14 (2013), pp. 1–19. URL: http://www.jstatsoft.org/v55/i14/ (see p. 16).

[17] Michael J. Kane, John Emerson, and Stephen Weston. "Scalable Strategies for Computing with Massive Data". In: *Journal of Statistical Software* 55.14 (2013), pp. 1–19. URL: http://www.jstatsoft.org/v55/i14/ (see p. 16).

[18] Hyunsoo Kim and Haesun Park. "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis." In: *Bioinformatics (Oxford, England)* 23.12 (2007), pp. 1495–502. ISSN: 1460-2059. DOI: 10.1093/bioinformatics/btm134. URL: http://www.ncbi.nlm.nih.gov/pubmed/17483501 (see pp. 4, 5, 11, 24, 25).

[19] Pierre L'Ecuyer, Richard Simard, and E.J. Chen. "An object-oriented random-number package with many long streams and substreams". In: *Operations Research* 50.6 (2002), pp. 1073–1075. URL: http://www.jstor.org/stable/3088626 (see p. 13).

[20] D D Lee and HS Seung. "Algorithms for non-negative matrix factorization". In: *Advances in neural information processing systems* (2001). URL: http://scholar.google.com/scholar?q=intitle:Algorithms+for+non-negative+matrix+factorization\#0 (see pp. 5, 12).

[21] J L Marchini, C Heaton, and B D Ripley. *fastICA: FastICA Algorithms to Perform ICA and Projection Pursuit*. R package version 1.2-2. 2019. URL: https://CRAN.R-project.org/package=fastICA (see p. 6).

[22] Microsoft and Steve Weston. *foreach: Provides Foreach Looping Construct*. R package version 1.4.8. 2020. URL: https://CRAN.R-project.org/package=foreach (see p. 16).

[23] Paul Murrell. *gridBase: Integration of base and grid graphics*. R package version 0.4-7. 2014. URL: https://CRAN.R-project.org/package=gridBase (see p. 24).

[24] Alberto Pascual-Montano, Jose Maria Carazo, K Kochi, D Lehmann, and R D Pascual-marqui. "Nonsmooth nonnegative matrix factorization (nsNMF)". In: *IEEE Trans. Pattern Anal. Mach. Intell* 28 (2006), pp. 403–415 (see pp. 5, 12, 28).

[25] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2011. URL: http://www.r-project.org (see p. 1).

[26] Jonathan Le Roux and Alain de Cheveigné. "Adaptive template matching with shift-invariant semi-NMF". In: *Science And Technology* (2008). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.143.6846&rep=rep1&type=pdf (see p. 27).

[27] Steve Weston. *doMPI: Foreach Parallel Adaptor for the Rmpi Package*. R package version 0.2.2. 2017. URL: https://CRAN.R-project.org/package=doMPI (see pp. 16, 17).

[28] Junying Zhang, Le Wei, Xuerong Feng, Zhen Ma, and Yue Wang. "Pattern expression nonnegative matrix factorization: algorithm and applications to blind source separation." In: *Computational intelligence and neuroscience* 2008 (2008), p. 168769. ISSN: 1687-5265. DOI: 10.1155/2008/168769. URL: http://www.ncbi.nlm.nih.gov/pubmed/18566689 (see pp. 5, 12).

```
# The cell type is used to label rows and columns
consensusmap(res.multirun, annCol=esGolub, tracks=NA)
```

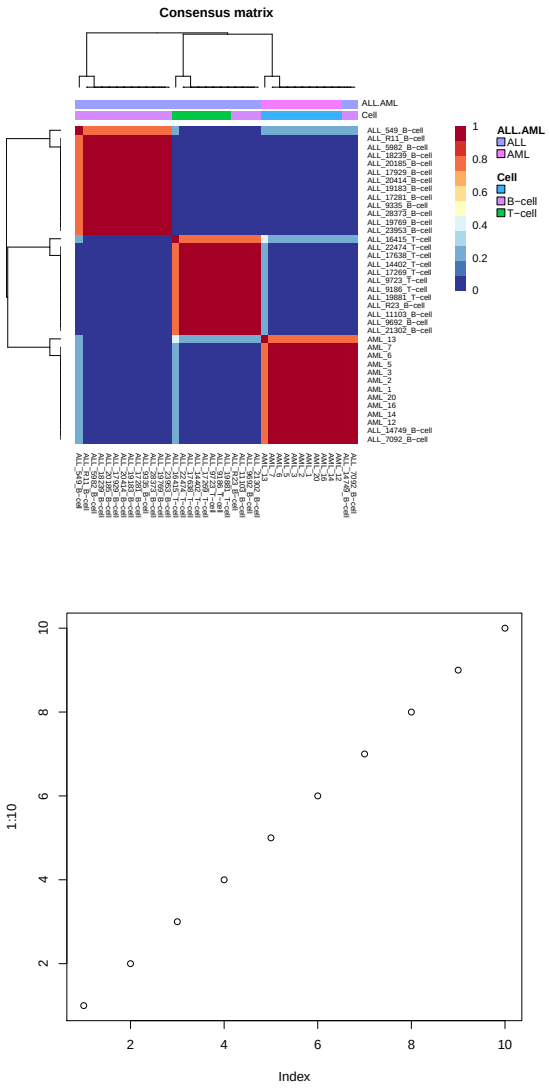

Figure 6: Heatmap of consensus matrices from 10 runs on the reduced dataset (left) and from 100 runs on the complete Golub dataset (right).