Using Package NMF

Renaud Gaujoux, <renaud@cbio.uct.ac.za>

June 16, 2010

This vignette presents the NMF package, which implements a framework for Nonnegative Matrix Factorization (NMF) algorithms in R [R Software, 2008]. The objective is to provide an implementation of some standard algorithms, while allowing the user to easily implement new methods that integrate into the package's framework.

Contents

The last stable version of the NMF package can be installed from any CRAN repository mirror, , and loaded with the standard calls:

```
## Not run
# install.packages('NMF')
library(NMF)
```

1 Overview

1.1 Package features

This section provides a quick overview of the `NMF` 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:

- 7 built-in algorithms;
- 4 built-in seeding methods;
- Single interface to perform all aglorithms, 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 based on 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 containned 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 equivalenty 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 [A. Cichocki *et al.*, 2004].

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.*, 2006, Chu *et al.*, 2004] and references therein.

The `NMF` package implements a number of published algorithms, and provides a general framework to implement other ones.

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()
```

Key	Description
`brunet`	Standard NMF. Based on Kullbach-Leibler divergence, it uses simple multiplicative updates from [Lee and Seung, 2000], enhanced to avoid numerical underflow. **Reference:** [Brunet *et al.*, 2004]
`lee`	Standard NMF. Based on euclidean distance, it uses simple multiplicative updates **Reference:** [Lee and Seung, 2000]
`nsNMF`	Nonsmooth NMF. Uses a modified version of Lee and Seung's multiplicative updates for Kullbach-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 L., 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. **Reference:** [Kim and Park, 2007]

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

```
[1] "brunet" "lee"    "nsNMF"  "offset" "pe-nmf" "snmf/l" "snmf/r"
# retrieve a specific algorithm: 'brunet'
nmfAlgorithm('brunet')
<object of class:  NMFStrategyIterative >
name:         brunet
objective:          'KL'
NMF model:          NMFstd
<Iterative schema:>
Update :  'nmf.update.brunet'
Stop :  'nmf.stop.consensus'
WrapNMF :  ''
# partial match is also fine
identical(nmfAlgorithm('br'), nmfAlgorithm('brunet'))
[1] TRUE
```

Table 1 gives a short description of each one of the built-in algorithms:

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[1]), 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.

[1]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 and Park, 2007].

Key	Description
`ica`	Uses the result of an Independent Component Analysis (ICA) (from the `fastICA` package). 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:** [?]
`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.

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 major drawback is that to achieve stability it requires 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 reasonnable starting point from the target matrix itself. The objective is to produce deterministic algorithms that need to run only once, still giving meaningful results.

For a review on some existing NMF initializations see [Albright *et al.*, 2006] and references therein.

The `NMF` package implements a number of already published seeding methods, and provides a general framework to implement other ones.

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

Table 2 gives a short description of each one of the built-in seeding methods:

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 is installed. The interface has four main parameters:

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

`x` is the target `matrix`, `data.frame` or `ExpressionSet` [2]

`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.

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 L., 2008]). Moreover they do not show up in the listing returned by the `nmfAlgorithm` function, unless argument `all=TRUE`:

```
nmfAlgorithm(all=TRUE)
 [1] "brunet"    "lee"       "nsNMF"     "offset"    "pe-nmf"    ".R#brunet"
 [7] ".R#lee"    ".R#nsNMF"  ".R#offset" "snmf/l"    "snmf/r"

# to get all the algorithms that have a secondary R version
nmfAlgorithm(type='R')
      brunet          lee        nsNMF       offset
 ".R#brunet"     ".R#lee"  ".R#nsNMF" ".R#offset"
```

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(type='R')
meth <- c(names(meth), meth)
meth
```

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

```
                                                              brunet          lee
        "brunet"        "lee"       "nsNMF"     "offset" ".R#brunet"     ".R#lee"
           nsNMF      offset
      ".R#nsNMF" ".R#offset"
# load the Golub data
data(esGolub)
# compute NMF for each method
res <- nmf(esGolub, 3, meth, seed=123456)
# extract only the elapsed time
t <- sapply(res, runtime)[3,]
```

	C	R	Speed.up
brunet	4.72	11.83	2.50
lee	7.90	11.65	1.47
nsNMF	8.04	16.13	2.01
offset	8.97	21.67	2.42

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

2 Use case: Golub dataset

We illustrate the functionalities and the usage of the `NMF` package on the – now standard – Golub dataset on leukemia. It was used in several papers on NMF [Brunet *et al.*, 2004, Gao and Church, 2005] and is included in the `NMF` 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 and varMetadata description:
         Sample: Sample name from the file ALL_AML_data.txt
         ALL.AML: ALL/AML status
         Cell: Cell type
     featureData
       featureNames: M12759_at, U46006_s_at, ..., D86976_at  (5000 total)
       fvarLabels and fvarMetadata description:
         Description: Short description of the gene
     experimentData: use 'experimentData(object)'
     Annotation:
esGolub <- esGolub[1:200,]
```

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
       distance metric:  'KL'
       residuals:  543535.7
       Iterations: 510
       Timing:
          user  system elapsed
         0.530   0.000   0.536
```

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        residuals
        3.000000e+00     6.392676e-01     6.217884e-01     5.435357e+05
               niter              cpu          cpu.all             nrun
        5.100000e+02     5.300000e-01     5.300000e-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        residuals            niter              cpu
        8.232656e-01     5.435357e+05     5.100000e+02     5.300000e-01
             cpu.all             nrun
        5.300000e-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        residuals            niter              cpu
        3.926954e-01     5.435357e+05     5.100000e+02     5.300000e-01
             cpu.all             nrun
        5.300000e-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 and Park, 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 0.0162700 0.0548900 0.1185000 0.1210000 1.0000000
# compute the scores and characterize each metagene
s <- extractFeatures(res)
str(s)
```

```
List of 3
 $ 1: int [1:8] 2 39 74 91 103 167 174 190
 $ 2: int [1:13] 1 8 25 26 41 42 59 64 69 94 ...
 $ 3: int [1:5] 43 120 128 129 130
 - attr(*, "threshold")= num 0.251
```

2.2 Specifying the algorithm

2.2.1 Built-in algorithms

The `NMF` 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" "lee"    "nsNMF"  "offset" "pe-nmf" "snmf/l" "snmf/r"
```

The algorithm used to compute the NMF is specified in the third argument (`method`). For example, to use the Lee and Seung [Lee and Seung, 2000] NMF algorithm 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
       distance metric:  <function>
       residuals:  67.35798
       parameters:
       $alpha
       [1] 0.01

       $beta
       [1] 1

       Iterations: 2000
       Timing:
          user  system elapsed
         1.820   0.000   1.819
```

2.2.2 Custom algorithms

The NMF 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 method

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] "ica"    "nndsvd" "none"   "random"
```

To use a specific method to seed the computation of a factorization, one can provide the name of the seeding method:

```r
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
       distance metric:  'KL'
       residuals:  547143.5
       Iterations: 1090
       Timing:
          user  system elapsed
         1.140   0.000   1.145
```

2.3.2 Numerical seed

Another possibility, useful when comparing methods or testing the reproducibility of the results, is to set the seed of the random generator by passing a numerical value in argument `seed`. This will call the function `set.seed` from package `base` before using the `'random'` seeding method:

```r
res <- nmf(esGolub, 3, seed=123456)
res
     <Object of class: NMFfit >
      # Model:
       <Object of class: NMFstd >
       features: 200
       basis/rank: 3
       samples: 38
      # Details:
       algorithm:  brunet
       seed:  123456
       distance metric:  'KL'
       residuals:  543535.7
       Iterations: 510
       Timing:
          user  system elapsed
         0.540   0.000   0.542
```

By default the value of the random seed is restored when the `nmf` function exits. This behaviour can be changed by specifying the option `restore.seed=FALSE` or `'-r'`.

2.3.3 Fixed factorization

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

```
n <- nrow(esGolub); p <- ncol(esGolub)
res <- nmf(esGolub, 3, seed=NULL, W=matrix(0.5, n, 3), H=matrix(0.3, 3, p))
res
     <Object of class: NMFfit >
      # Model:
       <Object of class: NMFstd >
       features: 200
       basis/rank: 3
       samples: 38
      # Details:
       algorithm:  brunet
       seed:  none
       distance metric:  'KL'
       residuals:  818694.4
       Iterations: 420
       Timing:
          user  system elapsed
         0.440   0.000   0.441
```

Important: in this case, argument `seed` must absolutely be set to `NULL`, otherwise the model instanciated with matrices W and H would only be used as a template, and reset passing it to the default seeding method.

Two alternative ways of doing this would be to pass matrices W and H through argument `model`, or a NMF model to argument `seed`:

```
res <- nmf(esGolub, 3, seed=NULL
                , model=list(W=matrix(0.5, n, 3), H=matrix(0.3, 3, p)))
# or
res <- nmf(esGolub, 3, seed=nmfModel(W=matrix(0.5, n, 3), H=matrix(0.3, 3, p)))
```

2.3.4 Custom function

The `NMF` 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
  Total timing:
   user  system elapsed
  1.450   0.100   2.494
```

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 inherit from 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
       Total timing:
        user  system elapsed
       4.460   0.130   2.613
       Sequential timing:
        user  system elapsed
        4.29    0.02    4.52
```

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

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[3] for the Golub gene expression matrix (5000×38) takes about 27096Kb[4].

2.4.1 Parallel computing on multi-core machines

To speed-up the analysis whenever possible, the `NMF` package implements transparent parallel computations when run on multi-core machines. It uses the `foreach` framework developed by REvolution Computing [foreach, 2009], together with the related `doMC` parallel backend [doMC, 2009] – based on the `multicore` package – to make use of all the CPUs available on the system. Each core will simultaneously perform part of the runs. Therefore, the required

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

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

memory increases linearly with the number of cores used. When only the best run is of interest, the memory usage is optimized by using shared memory and mutex objects from the `bigmemory` package, to only keep the current best factorization.

IMPORTANT NOTE: because it uses the `multicore` package, parallel computation over multi-cores is available only for Unix and Mac machines. The parallel computation is based on the `doMC` and `multicore` packages, so the same care should be taken as stated in the vignette of `doMC`:

> It is not safe to use doMC from R.app on MacOS 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.

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

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:

- 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 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).

Examples

The following exmaples are run with `.options='v'` which turn on verbosity. However *Sweave* do not show all the messages. The user is therefore encouraged 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')
     Runs: 2 1 4 3 5 ... DONE
     <Object of class: NMFfitX1 >
       Method: brunet
       Runs:  5
       Total timing:
        user  system elapsed
       4.060   0.150   2.348
```

```
# specifying the number of cores to use
nmf(esGolub, 3, nrun=5, .opt='v', .pbackend=2)
# force parallel computation: use option 'P'
nmf(esGolub, 3, nrun=5, .opt='vP')
```

2.4.2 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 to perform 100 factorizations using hybrid parallel computation on 4 quadri-core machines – i.e. making use of all the cores computation on each machine.

The scripts used to launch and run the factorizations can be found in file *mpi.R* in the package's *examples* directory:

```
file.show(file.system('examples/mpi.R', package='NMF'))
# and
file.show(file.system('examples/mpi_run.sh', package='NMF'))
```

The script file *mpi.R* contains some extra code to log and trace the computation. Reducing it to the essential gives the following piece of code:

```
## 0. Create and register an MPI cluster
library(doMPI)
cl <- startMPIcluster()
registerDoMPI(cl)
library(NMF)
## 1. Schedule the runs accross the workers
nrun <- 100;
nworker <- getDoParWorkers();
ntasks <- rep(round(nrun/nworker), nworker)
# allocate remainder runs
if( (remain <- nrun %% nworker) > 0 )
        ntasks[1:remain] <- ntasks[1:remain] + 1
## 2. Send the jobs to the workers using a foreach loop
t <- system.time({
        res <- foreach(i=1:getDoParWorkers(), n=ntasks,
                .packages = c('NMF', 'doMC', 'Biobase')) %dopar% {

                # each worker run its factorizations in parallel
                #Note: only the best result is kept
                data(esGolub)
                nmf(esGolub, 3, 'brunet', nrun=n, .opt='p')
        }
})
## 3. reduce the result and save it in a file
res <- NMF:::join(res, runtime.all=t)
```

```
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.

```
#!/bin/bash
#$ -cwd
#$ -q opteron.q
#$ -pe mpich 5
echo "Got $NSLOTS slots. $TMP/machines"

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

2.4.3 Forcing sequential execution

When running on a single core machine, NMF 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 2.4.1). 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 `'-p'` or by setting the parallel backend to `NULL`, `'seq'` or ■:

```
# force sequential computation by sapply: use option '-p' or .pbackend=''
nmf(esGolub, 3, nrun=5, .opt='v-p')
nmf(esGolub, 3, nrun=5, .opt='v', .pbackend='')
# or use the SEQ backend of foreach: .pbackend=NULL or 'seq'
nmf(esGolub, 3, nrun=5, .opt='v', .pbackend=NULL)
nmf(esGolub, 3, nrun=5, .opt='v', .pbackend='seq')
```

2.5 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.

The NMF package provides functions to run this procedure and plot the quality measures. Note that this can be lenghty. 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.

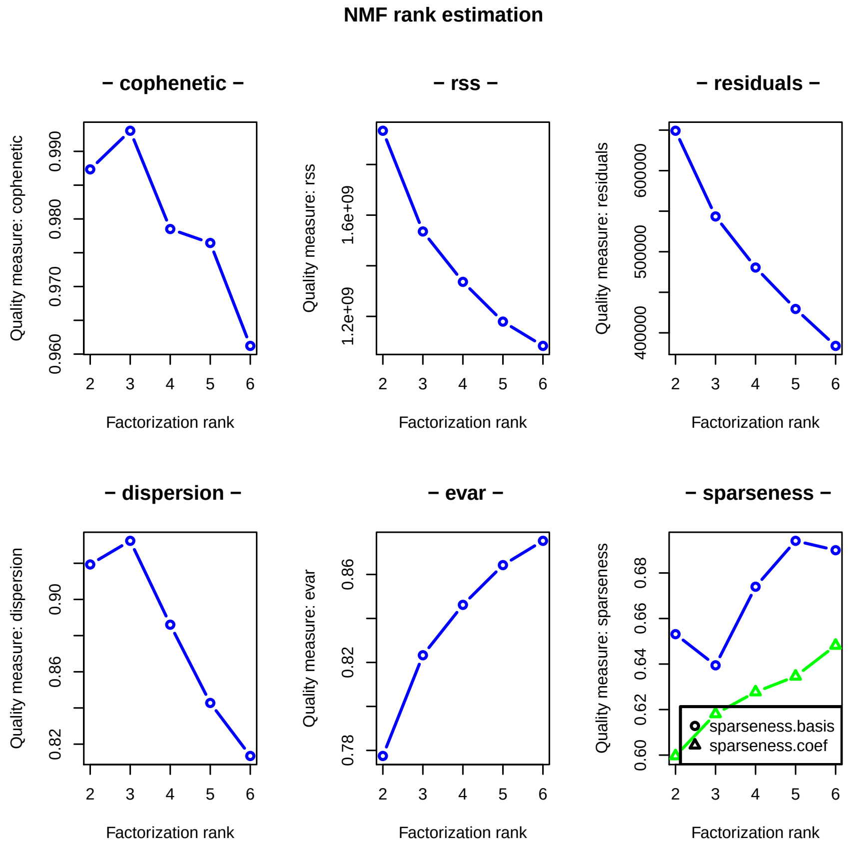

Figure 1: Estimation of the rank: Quality measures computed from 10 runs for each value of r

```
# perform 10 runs for each value of r in range 2:6
estim.r <- nmfEstimateRank(esGolub, range=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 by the following call, the result is shown in Figure 2.5:

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

Several approaches have 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 and Höglund, 2008] considered the smallest value at which the decrease in the RSS is lower than the decrease of the RSS obtained from random data.

2.5.1 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 and Höglund, 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 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 the "random" measures is easily added to the plot for comparison (see Figure 2.5.1):

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

2.6 Visualization methods

Error track

If the NMF computation is performed with error tracking enabled – using argument `.options` – the trajectory of the objective value can be plot with method `plot` (see Figure 3):

```
res <- nmf(esGolub, 3, .options='t')
# or alternatively:
# res <- nmf(esGolub, 3, .options=list(track=TRUE))
plot(res)
```

Heatmaps

Method `metaHeatmap` provides an easy way to vizualize the resulting metagenes, metaprofiles and, in the case of multiple runs, the consensus matrix. It produces pre-configured heatmaps based on function `heatmap.2` from package `gplots`. Examples of those heatmaps are shown in figures 4, 5, 6 and 7.

The following – default – call plots the metaprofiles matrix (see result Figure 4):

```
# default is to plot metaprofiles
metaHeatmap(res)
```

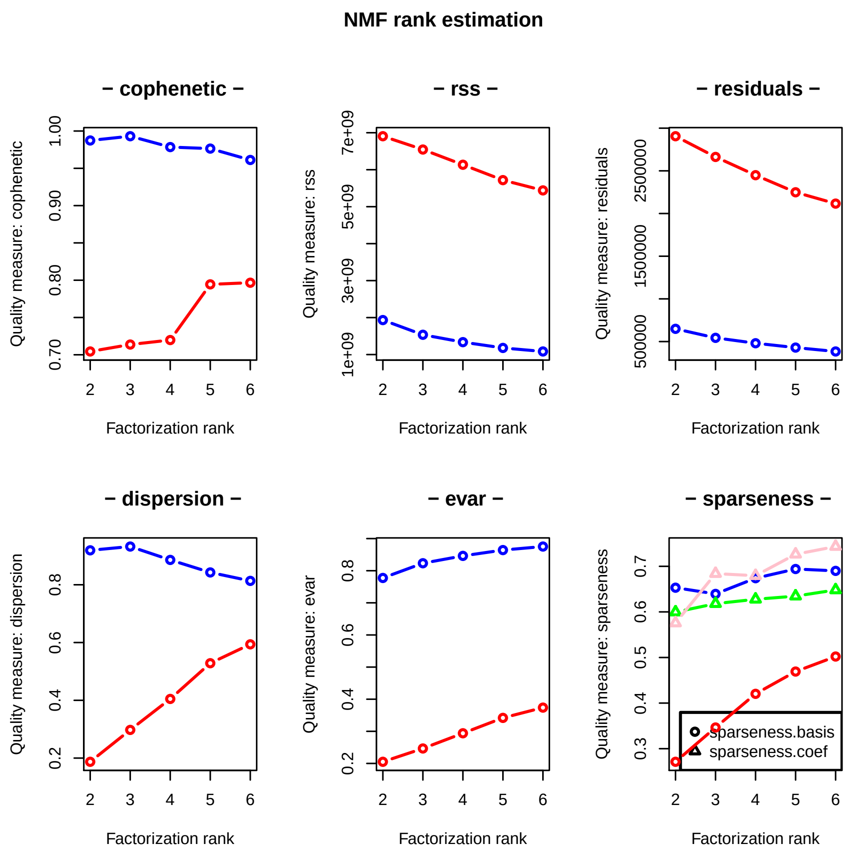

Figure 2: Estimation of the rank: Comparison of the quality measures with those obtained from randomized data. The curves for the actual data are in blue and green, those for the randomized data are in red and pink. The estimation is based on Brunet's algorithm.

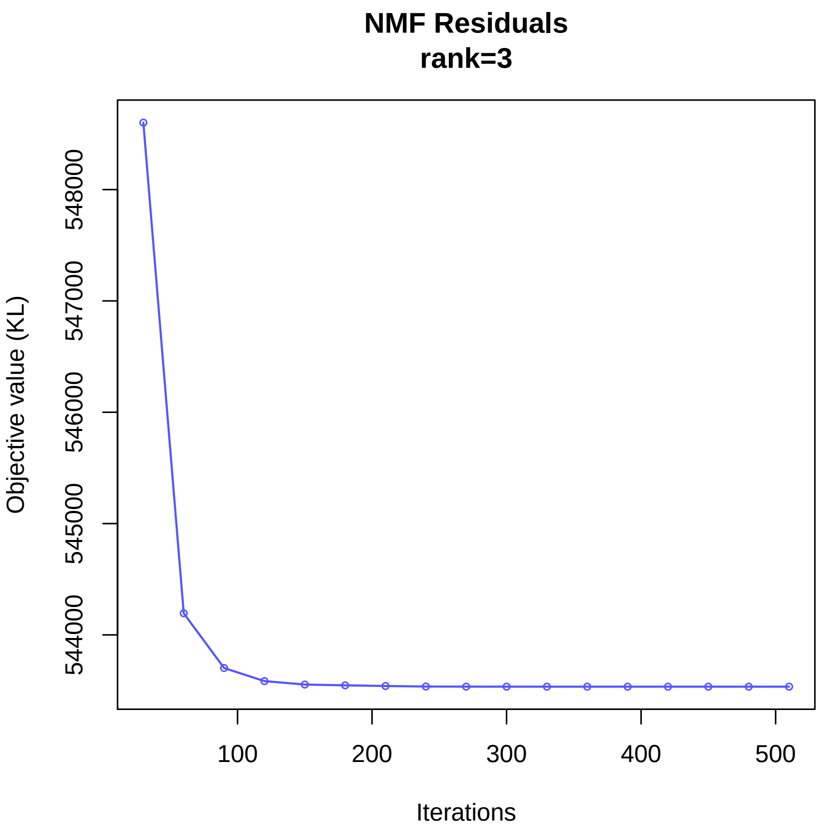

Figure 3: Error track for a single NMF run

The metagenes matrix can be plotted specifying the second argument `what` (see result Figure 5). We use argument `filter` to select only the genes that are specific to each metagene. With `filter=TRUE`, the selection method is the one described in [Kim and Park, 2007].

```
metaHeatmap(res, what='features', filter=TRUE)
```

In the case of multiple runs method `metaHeatmap` plots the consensus matrix, i.e. the average connecticity matrix accross the runs (see results Figures 6 and 7 for a consensus matrix obtained with 100 runs of Brunet's algorithm on Golub dataset):

```
# The cell type is used to label rows and columns
metaHeatmap(res.multirun, labRow=esGolub$Cell, labCol=esGolub$Cell)
```

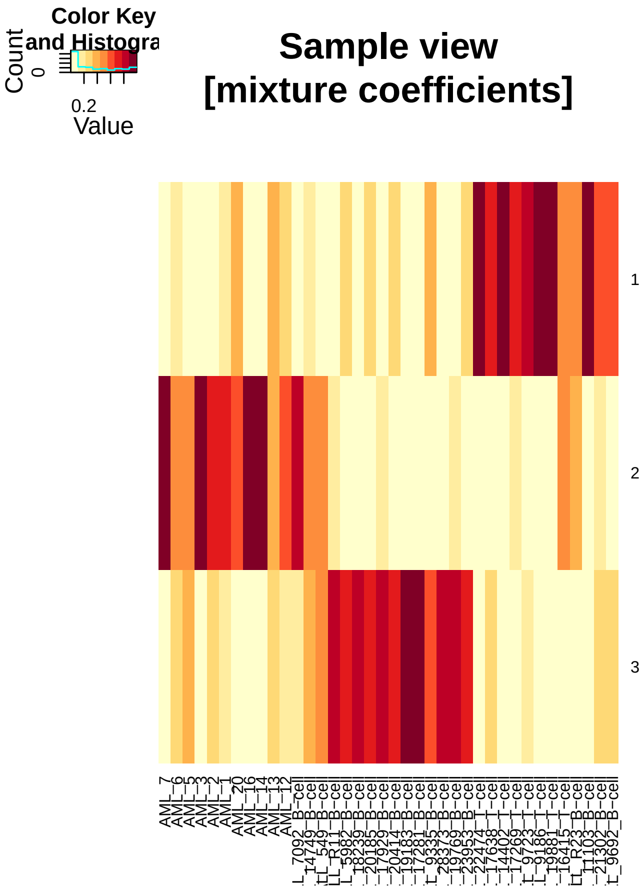

Figure 4: Heatmap of metaprofiles

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.

```
res.multi.method <- nmf(esGolub, 3, list('brunet', 'lee', 'ns'), seed=123456)
```

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      metric rank sparseness.basis sparseness.coef
     brunet brunet 123456        'KL'    3        0.6392676       0.6217884
     lee       lee 123456 'euclidean'    3        0.7268875       0.4465608
     nsNMF   nsNMF 123456        'KL'    3        0.6777185       0.7350386
              residuals niter  cpu cpu.all nrun
```

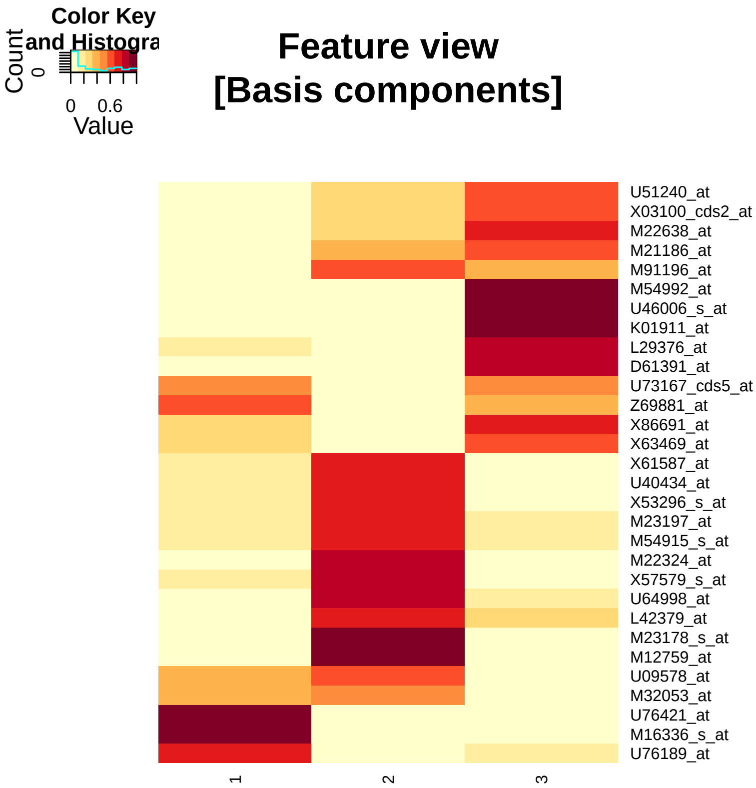

Figure 5: Heatmap of metagenes

```
brunet     543535.7    510 0.55     0.55    1
lee     673513120.5   1780 1.87     1.87    1
nsNMF      585106.4    970 1.37     1.37    1



       method   seed      metric rank sparseness.basis sparseness.coef
brunet brunet 123456        'KL'    3        0.6392676       0.6217884
lee       lee 123456 'euclidean'    3        0.7268875       0.4465608
nsNMF   nsNMF 123456        'KL'    3        0.6777185       0.7350386
          purity   entropy   residuals niter  cpu cpu.all nrun
brunet 0.8157895 0.3926954    543535.7   510 0.55    0.55    1
lee    0.5789474 0.7231282 673513120.5  1780 1.87    1.87    1
nsNMF  0.7894737 0.4691212    585106.4   970 1.37    1.37    1
```

When the computation is performed with error tracking enabled, an error plot is produced by method `plot` (see figure 8):

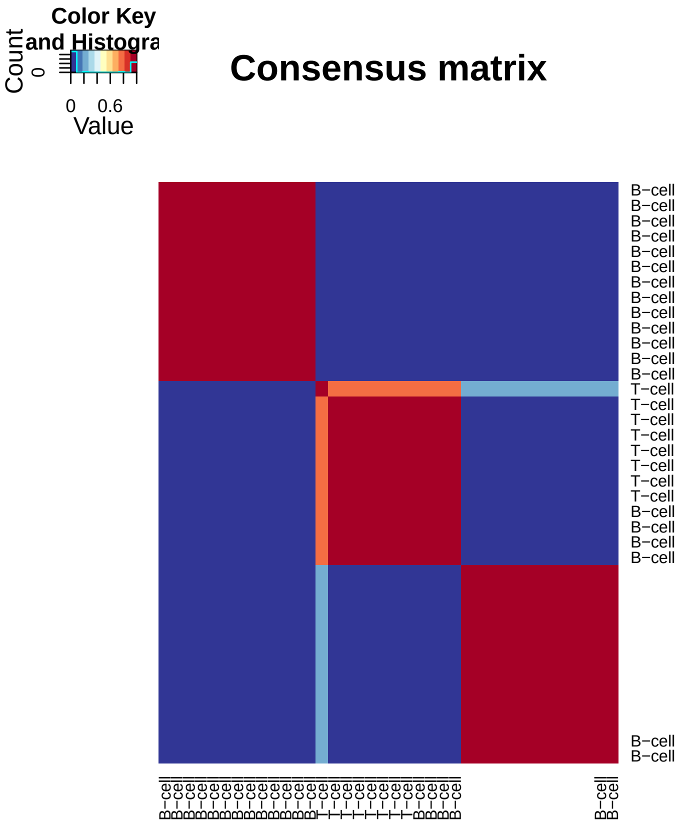

Figure 6: Heatmap of consensus matrix

```
res <- nmf(esGolub, 3, list('brunet', 'lee', 'ns'), seed=123456, .options='t')
plot(res)
```

3 Extending the package

We developed the `NMF` 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 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.

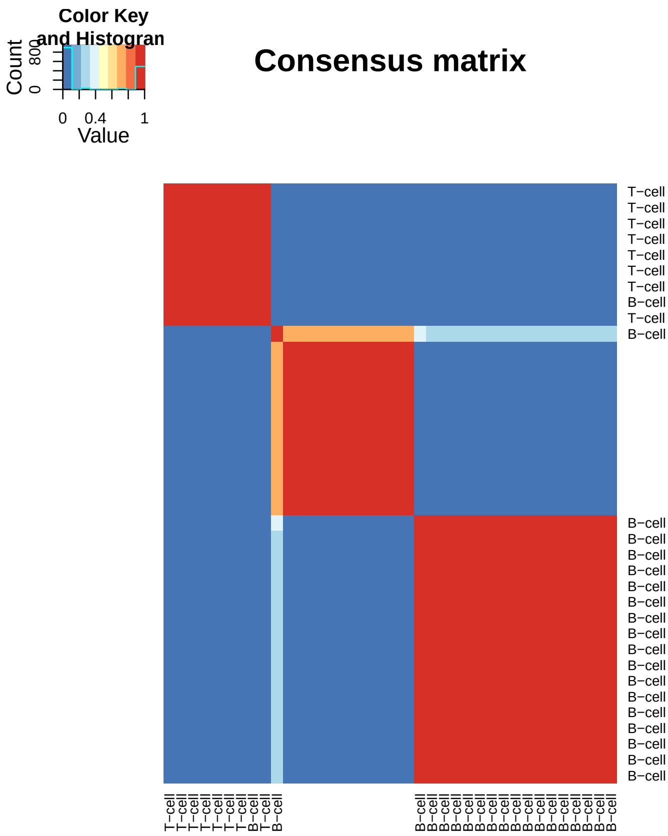

Figure 7: Heatmap of consensus matrix (100 runs of Brunet's algorithm on Golub dataset)

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...);

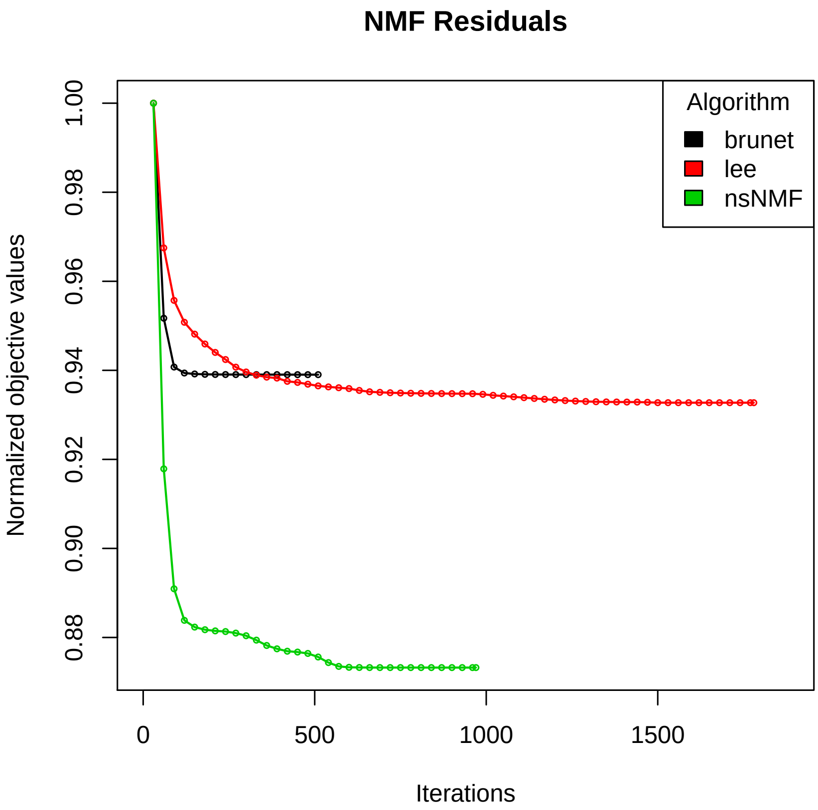

Figure 8: Error tracks comparing methods 'brunet', 'lee', 'nsNMF'

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)
        metagenes(fit(seed)) <- abs(pca$rotation[,1:factorization.rank])
        # use the rotated matrix to get the mixture coefficient
        # use a scaling factor (just to illustrate the use of extra parameters)
        metaprofiles(fit(seed)) <- t(abs(pca$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, noise=TRUE)
```

```
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.algo.88bd3ec
       seed:  random
       distance metric:  'euclidean'
       residuals:  637.5985
       parameters:
       $scale.factor
       [1] 10

       Timing:
          user  system elapsed
         0.000   0.000   0.031
```

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:

```
# based on Kullbach-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.algo.2e75443d
  seed:  random
  distance metric:  'KL'
  residuals:  1631.873
  parameters:
  $scale.factor
  [1] 10

  Timing:
     user  system elapsed
    0.000   0.000   0.003
```

```
<Object of class: NMFfit >
 # Model:
  <Object of class: NMFstd >
  features: 50
  basis/rank: 3
  samples: 20
 # Details:
  algorithm:  NMF.algo.2e7ccae3
  seed:  random
  distance metric:  <function>
  residuals:  9.356993
  parameters:
  $scale.factor
  [1] 10

  Timing:
     user  system elapsed
    0.000   0.000   0.003
```

3.1.3 Defining algorithms for mixed sign data

All the algorithms implemented in the NMF 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 [Chris Ding *et al.*, 2008, Le Roux *et al.*, 2008]). Strictly speaking these methods do not 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 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 [5]. To

[5]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.

extend the capabilities of the NMF 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[6]:

```
# put some negative input data
V.neg <- V; V.neg[1,] <- -1;
# this generates an error
err <- try( nmf(V.neg, 3, my.algorithm, scale.factor=10) )
err
     [1] "Error in .local(x, rank, method, ...) : \n  Input matrix x contains some negative entries.\n"
     attr(,"class")
     [1] "try-error"

# 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.algo.43c518b6
       seed:  random
       distance metric:  'euclidean'
       residuals:  639.6262
       parameters:
       $scale.factor
       [1] 10

       Timing:
          user  system elapsed
         0.010   0.000   0.003
```

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 retireved calling the `nmfModel()` function with no argument:

```
nmfModel()
     [1] "NMFstd"    "NMFOffset" "NMFns"
```

[6] 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.

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 L., 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)
        metagenes(data.model) <- abs(pca$rotation[,1:factorization.rank])
        # use the rotated matrix to get the mixture coefficient
        # use a scaling factor (just to illustrate the use of extra parameters)
        metaprofiles(data.model) <- t(abs(pca$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: [ 0.6761489 0.5007902 0.848937 0.3704305 0.8774585 ... ]
      # Details:
       algorithm:  NMF.algo.3e256674
       seed:  random
       distance metric:  'euclidean'
       residuals:  353.7908
       parameters:
       $scale.factor
```

```
[1] 10

Timing:
   user  system elapsed
  0.000   0.000   0.003
```

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)))
        metagenes(model) <- target[,order(w.cols)[1:nbasis(model)]]

        # initialize H randomly
        metaprofiles(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.c699814
       distance metric:  'euclidean'
       residuals:  154.3545
       Iterations: 80
       Timing:
          user  system elapsed
         0.460   0.000   0.472
```

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

Currently the following options are available:

Option	Default value	Description
`default.algorithm`	brunet	Default NMF algorithm used by the `nmf` function when argument `method` is missing. The value should the key of one of the available NMF algorithms. See `?nmfAlgorithm`.
`track.interval`	30	Number of iterations between two points in the residual track. This option is relevant only when residual tracking is enabled. See `?nmf`.
`error.track`	FALSE	Toggle default residual tracking. When `TRUE`, the `nmf` function compute and store the residual track in the result – if not otherwise specified in argument `.options`. Note that tracking may significantly slow down the computations.
`default.seed`	random	Default seeding method used by the `nmf` function when argument `seed` is missing. The value should the key of one of the available seeding methods. See `?nmfSeed`.
`parallel.backend`	mc	Default parallel backend used used by the `nmf` function when argument `.pbackend` is missing. Currently the following values are supported: `'mc'` for multicore, `'seq'` for sequential, ■ for `sapply`.
`verbose`	FALSE	Toggle verbosity.
`debug`	FALSE	Toggle debug mode, which is an extended verbose mode.

5 Session Info

```
R version 2.11.1 (2010-05-31)
x86_64-pc-linux-gnu
```

```
locale:
 [1] LC_CTYPE=en_ZA.utf8       LC_NUMERIC=C
 [3] LC_TIME=en_ZA.utf8        LC_COLLATE=en_ZA.utf8
 [5] LC_MONETARY=C             LC_MESSAGES=en_ZA.utf8
 [7] LC_PAPER=en_ZA.utf8       LC_NAME=C
 [9] LC_ADDRESS=C              LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_ZA.utf8 LC_IDENTIFICATION=C

attached base packages:
[1] tools     stats     graphics  grDevices utils     datasets  methods
[8] base

other attached packages:
 [1] RColorBrewer_1.0-2  synchronicity_1.0.8 bigmemory_4.2.3
 [4] doMC_1.2.1          multicore_0.1-3     foreach_1.3.0
 [7] codetools_0.2-2     iterators_1.0.3     xtable_1.5-6
[10] NMF_0.4.5           Biobase_2.8.0
```

References

[Albright *et al.*, 2006] R. Albright, J. Cox, D. Duling, A. Langville, C. Meyer (2006). Algorithms, initializations, and convergence for the nonnegative matrix factorization. *NCSU Technical Report Math 81706*. http://meyer.math.ncsu.edu/Meyer/Abstracts/Publications.html.

[Badea L., 2008] Badea L. (2008). Profiles Common to Colon and Pancreatic Adenocarcinoma Using Simultaneous nonNegative. In *Pacific Symposium on Biocomputing*, *Volume 290* 2008:279–290.

[Berry *et al.*, 2006] Berry et al. (2006). Algorithms and Applications for Approximate Nonnegative Matrix Factorization. *Comput. Stat. Data Anal.*

[Brunet *et al.*, 2004] Brunet, J.~P., Tamayo, P., Golub, T.~R., and Mesirov, J.~P. (2004). Metagenes and molecular pattern discovery using matrix factorization. *Proc Natl Acad Sci U S A*, **101**(12), 4164–4169.

[A. Cichocki *et al.*, 2004] Andrzej Cichocki , Rafal Zdunek, and Shun-ichi Amari (2004). New algorithms For Non-negative Matrix Factorization In Application To Blind Source Separation.

[Chu *et al.*, 2004] M.T. Chu, F. Diele, R. Plemmons, S. Ragni. Optimality, computation, and interpretation of nonnegative matrix factorizations. *Technical Report*, Departments of Mathematics and Computer Science, Wake Forest University, USA.

[Chris Ding *et al.*, 2008] Ding, C., Li, T., Jordan, M.~I.. Convex and Semi-Nonnegative Matrix Factorizations *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2008, vol. 32, no. 1, pp. 45-55 [http://doi.ieeecomputersociety.org/10.1109/TPAMI.2008.277]

[doMC, 2009] REvolution Computing: *doMC: Foreach parallel adaptor for the multicore package* 2009, [http://CRAN.R-project.org/package=doMC]. [R package version 1.2.0].

[foreach, 2009] REvolution Computing: *doMC: Foreach parallel adaptor for the multicore package* 2009, [http://CRAN.R-project.org/package=doMC]. [R package version 1.3.0].

[Frigyesi and Höglund, 2008] Frigyesi A, Höglund M: **Non-negative matrix factorization for the analysis of complex gene expression data: identification of clinically relevant tumor subtypes.** *Cancer informatics* 2008, **6**:275–292, [http://view.ncbi.nlm.nih.gov/pubmed/19259414].

[Gao and Church, 2005] Gao Y, Church G: **Improving molecular cancer class discovery through sparse non-negative matrix factorization**. *Bioinformatics* 2005, **21**(21):3970–3975, [http://dx.doi.org/10.1093/bioinformatics/bti653].

[Gentleman *et al.*, 2004] Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J, Hornik K, Hothorn T, Huber W, Iacus S, Irizarry R, Leisch F, Li C, Maechler M, Rossini AJ, Sawitzki G, Smith C, Smyth G, Tierney L, Yang JYH, Zhang J: **Bioconductor: open software development for computational biology and bioinformatics.** *Genome biology* 2004, **5**:R80, [http://www.ncbi.nlm.nih.gov/pubmed/15461798].

[Hutchins *et al.*, 2008] Hutchins LNN, Murphy SMM, Singh P, Graber JHH: **Position-Dependent Motif Characterization Using Nonnegative Matrix Factorization.** *Bioinformatics (Oxford, England)* 2008, [http://view.ncbi.nlm.nih.gov/pubmed/18852176].

[Kim and Park, 2007] Kim H, Park H: **Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis.** *Bioinformatics (Oxford, England)* 2007, **23**:1495–502, [http://www.ncbi.nlm.nih.gov/pubmed/17483501].

[Le Roux *et al.*, 2008] Le Roux, J., de CheveignÃl', A., Parra, L.˜C. Adaptive Template Matching with Shift-Invariant Semi-NMF. In *NIPS* 2008, 921-928. [http://dblp.uni-trier.de/db/conf/nips/nips2008.html#RouxCP08]

[Lee and Seung, 2000] Lee, D.˜D. and Seung, H.˜S. (2000). Algorithms for non-negative matrix factorization. In *NIPS*, 556-562.

[Pascual-Montano *et al.*, 2006] Pascual-Montano, A., Carazo, J.˜M., Kochi, K., Lehmann, D., and Pascual-Marqui, R.˜D. (2006). Nonsmooth nonnegative matrix factorization (nsnmf). *IEEE transactions on pattern analysis and machine intelligence*, **28**(3), 403-415.

[R Software, 2008] R Development Core Team. R: A Language and Environment for Statistical Computing. Vienna, Austria. ISBN 3-900051-07-0. http://www.R-project.org.

[Zhang *et al.*, 2008] Zhang J, Wei L, Feng X, Ma Z, Wang Y: **Pattern expression nonnegative matrix factorization: algorithm and applications to blind source separation.** *Computational intelligence and neuroscience* 2008, [http://view.ncbi.nlm.nih.gov/pubmed/18566689].