Using Package NMF

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

January 6, 2010

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

The last stable version of the NMF package can be installed from any CRAN repository mirror via:

```
> install.packages('NMF')
```

It loads with the following standard call:

```
> library(NMF)
```

Contents

1 Overview of Nonnegative Matrix Factorization

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 $X \approx WH$, 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:

$$\min_{W,H \geq 0} \underbrace{[D(X, WH) + R(W, H)]}_{=F(W,H)} \tag{1}$$

where

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

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

  or the generalized Kullback-Leibler divergence.

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

- 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].

1.1 Algorithms

Algorithms to solve problem (1) iteratively build a sequence of matrices (W_k, H_k) that reduces at each step the value of the objective function F They differ in the optimization techniques used to compute updates for (W_k, H_k).

For reviews on NMF algorithms see [Berry *et al.*, 2006, Chu *et al.*, 2004] and references therein.

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

Implemented NMF algorithms are listed or retrieved with function `nmfAlgorithm`. Specific algorithms are retrieved by their name (a `character` key) that is partially matched against the list of available algorithms:

```
> # list all available algorithms
> nmfAlgorithm()
  [1] "brunet" "lee"    "lnmf"   "nsNMF"  "offset" "snmf/l" "snmf/r"
> # retrieve a specific algorithm: 'brunet'
> nmfAlgorithm('brunet')
```

```
  <object of class:  NMFStrategyIterative >
  name:         brunet
  objective:          'KL'
  NMF model:          NMFstd
  <Iterative schema:>
  Preprocess :  ''
  Update :  'nmf.update.brunet'
  Stop :  'nmf.stop.consensus'
  WrapNMF :  ''
> # partial match is also fine
> identical(nmfAlgorithm('br'), nmfAlgorithm('brunet'))
  [1] TRUE
```

1.2 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.

The more common seeding method is to start with a random guess, where the entries of W and/or H are drawn from a uniform distribution. 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 NMF algorithms' running time.

To tackle this problem, some methods have been proposed so as to compute a 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.

Package NMF implements a number of standard seeding methods, and provides a general framework to implement other ones.

Implemented seeding methods are listed or retrieved with function `nmfSeed`. Specific seeding methods are retrieved by their 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
```

[1] Some algorithms only need one matrix factor (either W or H) to be initialized. See for example SNMF algorithms.

1.3 How to run NMF algorithms

Method `nmf` provides a single interface to run NMF algorithms. It can perform NMF on object of class `matrix`, `data.frame` and `ExpressionSet`. The interface takes four main parameters:

```
nmf(x, rank, method='brunet', seed='random', ...)
```

- `x` is the target `matrix` – `data.frame` or `ExpressionSet`
- `rank` is the factorization rank
- `method` is the algorithm used to estimate the factorization. Default algorithm is from [Brunet *et al.*, 2004].
- `seed` is the seeding method used to compute the starting point. Default is to use a random initialization.

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

2 Use case: Golub dataset

The Golub dataset on leukemia used in [Brunet *et al.*, 2004] is included in package NMF. It is wrapped into an `ExpressionSet` object and can be loaded as follows. For performance reason we only use the first 1000 genes:

```
> data(esGolub)
> esGolub
  ExpressionSet (storageMode: lockedEnvironment)
  assayData: 5000 features, 38 samples
    element names: exprs
  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[1:1000,]
```

Note: To run this example, the `Biobase` package from BioConductor should be installed.

2.1 Single run

2.1.1 Performing a single run

The following code runs the default NMF algorithm on data `esGolub` with factorization rank equal to 3:

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

2.1.2 Handling the result

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

```
> res
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 1000
    basis/rank: 3
    samples: 38
   # Details:
    algorithm:  brunet
    seed:  random
    distance metric:  'KL'
    residuals:  2844567
    Timing:
       user  system elapsed
      2.688   0.080   2.801
```

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

```
> fit(res)
  <Object of class: NMFstd >
  features: 1000
  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] 1000   38
```

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

```
> summary(res)
         rank   sparseness         time    residuals
  3.000000e+00 6.731665e-01 2.688000e+00 2.844567e+06
```

If there is some prior knowledge of classes present in the data, extra measures about the unsupervised clustering's performance are be computed. Here we use the phenotypic variable `Cell` that gives the samples' cell-types (T-cell, B-cell or `NA`):

```
> summary(res, class=esGolub$Cell)
         rank   sparseness       purity      entropy         time    residuals
  3.000000e+00 6.731665e-01 9.210526e-01 1.543928e-01 2.688000e+00 2.844567e+06
```

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] 1000    3
> # get matrix H
> h <- coef(res)
> dim(h)
  [1]  3 38
```

If one wants to keep only part of the result, one can directly subset on the `NMF` object on features and samples (separately or simultaneously):

```
> # keep only the first 100 features
> res[1:100,]
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 100
    basis/rank: 3
    samples: 38
   # Details:
    algorithm:  brunet
    seed:  random
    distance metric:  'KL'
    residuals:  2844567
    Timing:
       user  system elapsed
      2.688   0.080   2.801
> # keep only the first 10 samples
> res[,1:10]
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 1000
    basis/rank: 3
    samples: 10
```

```
    # Details:
     algorithm:  brunet
     seed:  random
     distance metric:  'KL'
     residuals:  2844567
     Timing:
        user  system elapsed
       2.688   0.080   2.801
> # subset both features and samples:
> dim(res[1:100,1:10])
  [1] 100  10   3
```

2.1.3 Selecting the 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.
  5.776e-05 1.772e-02 5.554e-02 1.074e-01 1.367e-01 9.607e-01
> # compute the scores and characterize each metagene
> s <- extractFeatures(res)
> str(s)
  List of 3
   $ 1: int [1:44] 2 4 14 39 74 91 120 167 190 227 ...
   $ 2: int [1:30] 43 69 128 130 236 240 247 249 258 289 ...
   $ 3: int [1:37] 8 27 41 42 59 64 70 94 112 182 ...
   - attr(*, "threshold")= num 0.261
```

2.2 Specifying the algorithm

The algorithm used to compute the NMF is specified in the third argument (`method`). For example, 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)
> res
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFns >
    features: 1000
```

```
 basis/rank: 3
 samples: 38
 theta: 0.7
# Details:
 algorithm:  nsNMF
 seed:  random
 distance metric:  'KL'
 residuals:  3314535
 Timing:
    user  system elapsed
   5.976   0.004   6.098
```

2.3 Multiple runs

The default seeding method being random seeding, multiple runs are required to achieve stability. This can be done by setting argument `nrun` to the desired value. For performance reason we use `nrun=5` here, but a reasonnable choice would typically lies between 100 and 200:

```
> res.multirun <- nmf(esGolub, 3, nrun=5)
> res.multirun
 <Object of class: NMFSet >
 method: brunet
 runs: 5
 fits: 1
 Timing:
    user  system elapsed
  13.877   0.048  13.996
 Avg. timing:
    user  system elapsed
  2.7754  0.0096  2.7992
```

As we can see from the results above, the returned object contains only one fit, from the 5 runs that was perfomed. The default behaviour is to only keep the factorization achieving the lowest approximation error (i.e. the lowest objective value). However if one is interested in keeping the results from all the runs, one can set the option `keep.all=TRUE`:

```
> # using letter code 'k' in argument .options
> nmf(esGolub, 3, nrun=5, .options='k')
> # or explicitly setting the option
> nmf(esGolub, 3, nrun=5, .options=list(keep.all=TRUE))
```

2.4 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:

```
> res <- nmf(esGolub, 3, seed='nndsvd')
> res
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 1000
    basis/rank: 3
    samples: 38
   # Details:
    algorithm:  brunet
    seed:  nndsvd
    distance metric:  'KL'
    residuals:  2848368
    Timing:
       user  system elapsed
      6.232   0.012   6.250
```

Another possibility, useful when comparing methods, is to set the seed of the random generator passing a numerical value in argument `seed`. In this case, function `set.seed` from package `base` is called before using seeding method `'random'`:

```
> res <- nmf(esGolub, 3, seed=123456)
> res
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 1000
    basis/rank: 3
    samples: 38
   # Details:
    algorithm:  brunet
    seed:  123456
    distance metric:  'KL'
    residuals:  2844567
    Timing:
       user  system elapsed
      2.749   0.000   2.746
```

2.5 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 `errorPlot` (see Figure 1):

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

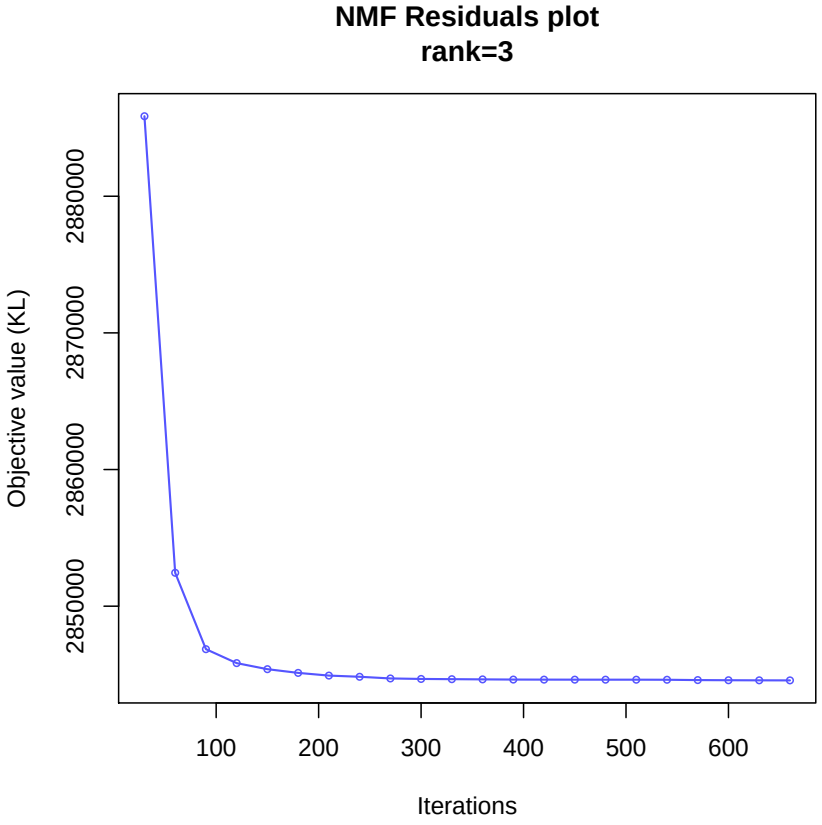

Figure 1: Error track for a single NMF run

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 2, 3, 4 and 5.

The following – default – call plots the metaprofiles matrix (see result Figure 2):

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

The metagenes matrix can be plotted specifying the second argument `what` (see result Figure 3). 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)
```

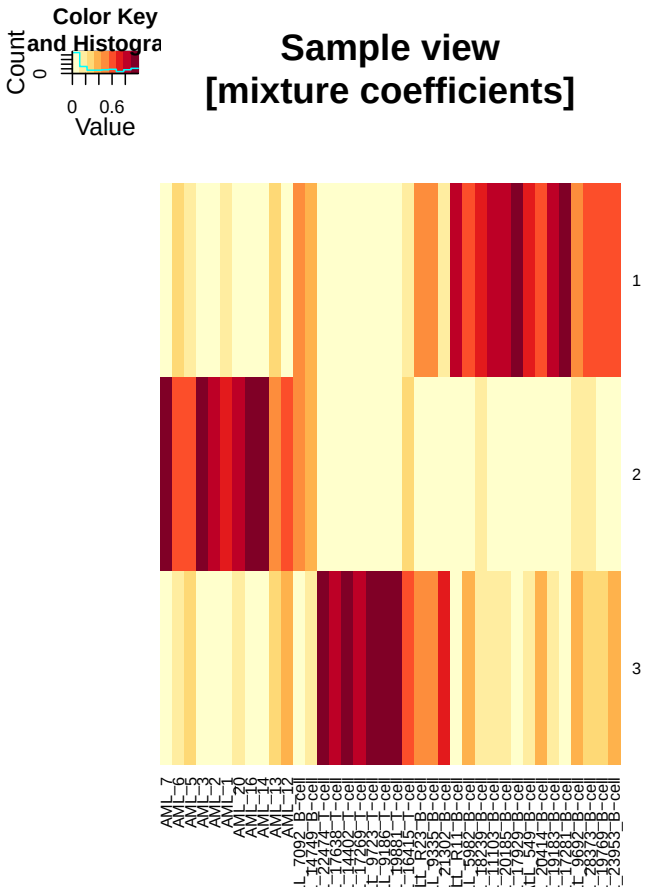

Figure 2: Heatmap of metaprofiles

In the case of multiple runs method `metaHeatmap` plots the consensus matrix, i.e. the average connecticity matrix accross the runs (see results Figures 4 and 5 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)
```

2.6 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:

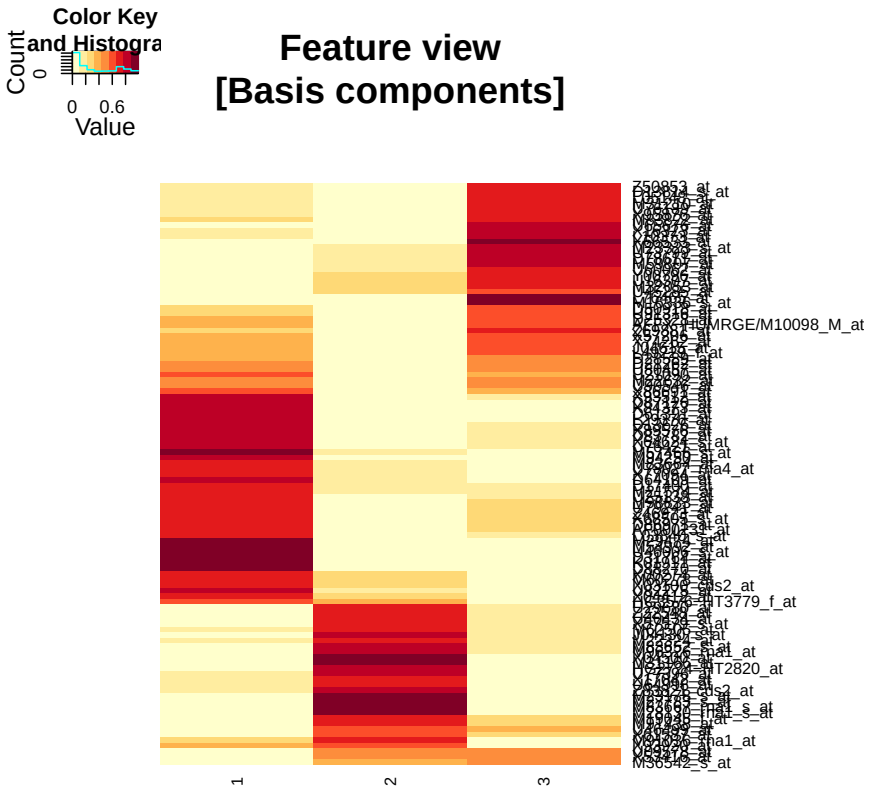

Figure 3: Heatmap of metagenes

```
> compare(res.multi.method)
        method   seed      metric rank sparseness  time  residuals
  brunet brunet 123456        'KL'    3  0.6731665 2.421    2844567
  nsNMF   nsNMF 123456        'KL'    3  0.7054783 4.748    3056230
  lee       lee 123456 'euclidean'    3  0.7013150 3.272 5850132039
> # If prior knowledge of classes is available
> compare(res.multi.method, class=esGolub$Cell)
        method   seed      metric rank sparseness    purity   entropy  time
  brunet brunet 123456        'KL'    3  0.6731665 0.9210526 0.1543928 2.421
  nsNMF   nsNMF 123456        'KL'    3  0.7054783 0.8947368 0.2368421 4.748
  lee       lee 123456 'euclidean'    3  0.7013150 0.7631579 0.4139661 3.272
         residuals
  brunet    2844567
  nsNMF     3056230
  lee    5850132039
```

When the computation is performed with error tracking enabled, an error plot is produced by method `errorplot` (see figure 6):

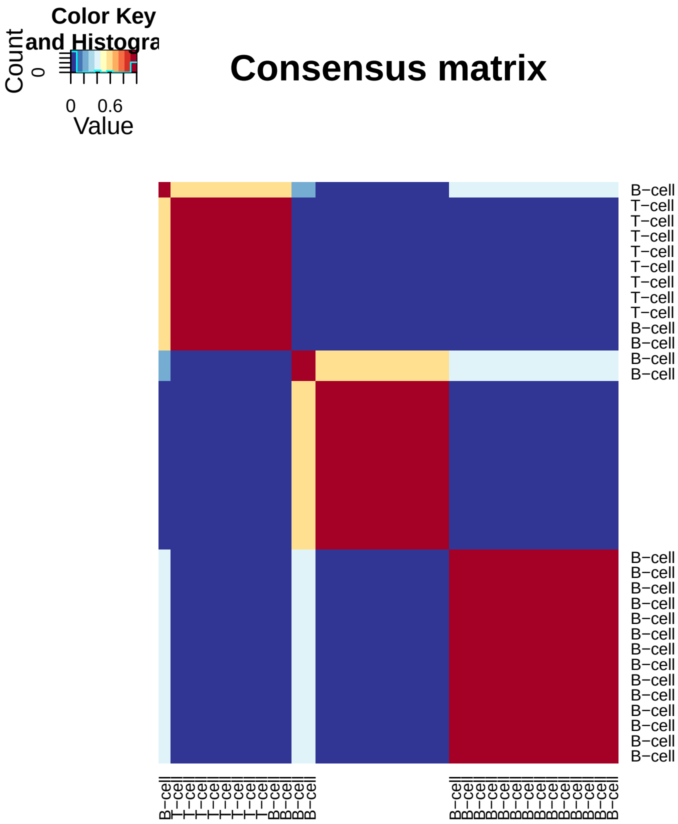

Figure 4: Heatmap of consensus matrix

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

3 Advanced usage

We developped package NMF with the objective to allow 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.

3.1 Custom algorithm

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

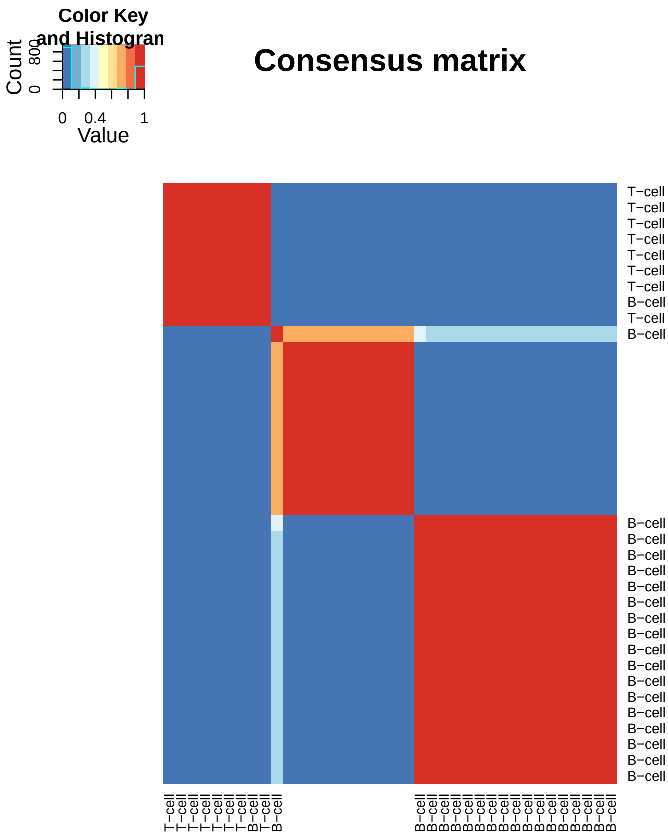

Figure 5: Heatmap of consensus matrix (100 runs of Brunet's algorithm on Golub dataset)

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

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

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:

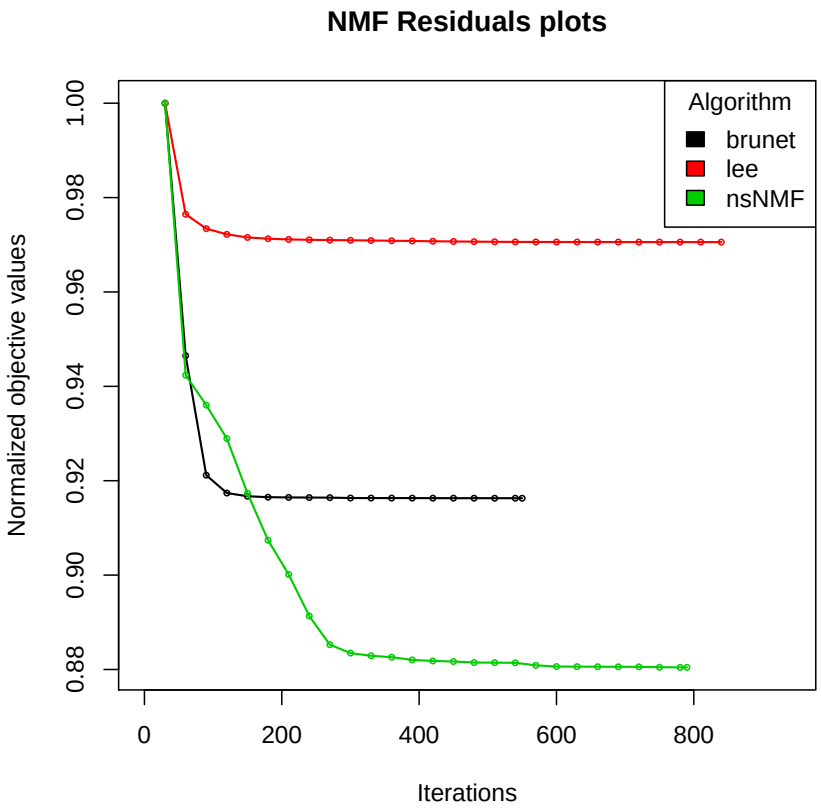

Figure 6: Error tracks comparing methods `'brunet', 'lee', 'nsNMF'`

```
> my.algorithm <- function(target, start, scale.factor=1){
          # do something with starting point
          # ...
          # for example:
          # 1. compute principal components
          pca <- prcomp(t(target), retx=TRUE)
          # 2. use the absolute values of the first PC for the metagenes
          # Note: the factorization rank is stored in object 'start'
          factorization.rank <- nbasis(start)
          metagenes(fit(start)) <- abs(pca$rotation[,1:factorization.rank])
          # use the rotated matrix to get the mixture coefficient
          # use a scaling factor (just to illustrate the use of extra parameters)
          metaprofiles(fit(start)) <- t(abs(pca$x[,1:factorization.rank])) / scale.factor

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

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.419ac241
    seed:  random
    distance metric:  'euclidean'
    residuals:  715.1727
    parameters:
    $scale.factor
    [1] 10

    Timing:
       user  system elapsed
      0.004   0.000   0.005
```

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 argumentobjective, 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.440badfc
    seed:  random
    distance metric:  'KL'
    residuals:  1638.295
    parameters:
    $scale.factor
    [1] 10

    Timing:
```

```
       user  system elapsed
      0.000   0.000   0.003
> # based on custom distance metric
> nmf(V, 3, my.algorithm, scale.factor=10
         , objective=function(target, x){
                         ( sum( (target-fitted(x))^4 ) )^{1/4}
                 }
 )
  <Object of class: NMFfit >
   # Model:
    <Object of class: NMFstd >
    features: 50
    basis/rank: 3
    samples: 20
   # Details:
    algorithm:  NMF.algo.3804823e
    seed:  random
    distance metric:  <function>
    residuals:  10.20292
    parameters:
    $scale.factor
    [1] 10

    Timing:
       user  system elapsed
      0.004   0.000   0.002
```

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.5c482a97
    distance metric:  'euclidean'
    residuals:  155.4585
    Iterations: 90
    Timing:
       user  system elapsed
      0.564   0.000   0.564
```

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.

[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.

[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].

[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.