Documentation: NMFIF, a Non-negative Matrix Factorization Integrative Framework

Thomas Gaudelet

Contents

1 Introduction

1.1 Data integration

With the advent of modern large-scale sequencing technologies, the availability of biomedical, multi-omics data has surged. Large repositories, such as BioGrid [3] and Reactome [7], make publicly available state-of-the art knowledge on the intricate cellular mechanisms for multiple organisms. Projects such as International Cancer Genome Consortium (ICGC, [22]) and Gene Expression Omnibus (GEO, [6]) collect various patient medical data from studies across the literature and make it available allowing growing number of studies into the biology of diseases.

Each omic data gives different insights into the state of a biological system. Furthermore, a crucial characteristic of biological data is the underlying inherent interactions among molecular entities that are the object of study at the core of system biology. Through these observations, many researchers have turned to integrative analysis with the objective of harnessing the information contained in the various type of biological data. This has proved an efficient strategy in a number of applications as exemplified by the numerous literature reviews on the subject [23, 19, 18, 10].

1.2 Non-negative Matrix Factorizations

Non-negative matrix factorization approaches have been used extensively both as a mean to integrate heterogenous data and reduce data dimensionality. Various versions of non-negative matrix factorization have been developed to accommodate different context [11, 13, 5]. They encompass all methods that decompose a matrix, representing relational links between two set of entities, into the product of smaller positive matrices, or factors, whose sizes control the degree of dimensionality reduction [20]. Crucially, one can derive an embedding in an unspecified latent space for each entity. Matrix factorization approaches have found success in numerous applications, including collaborative filtering [15] and biological data integration [9, 12, 8, 21, 2, 14].

1.3 Package description

We develop a C++ package, NMFIF, that allows for jointly optimizing multiple NMF objectives. Our software can flexibly combine four different types of NMF: standard NMF, Symmetric NMF (SNMF), Non-negative Matrix Factorizations (NMTF), and Symmetric Non-negative Matrix Factorizations (SNMTF) (see details in Section 3.1). Users use the NMFIF package to frame any joint NMF-based optimization problems without having to code the optimization procedure. To the best of our knowledge, this is the first software that allows this using update rules specifically derived for NMF optimizations.

2 Installation

2.1 Dependencies

NMFIF relies on some external libraries

- Armadillo (available at `http://arma.sourceforge.net/download.html`)
- HDF5 (available at `https://www.hdfgroup.org/downloads`)
- Lapack (available at `http://www.netlib.org/lapack/`)
- OpenBlas or MKL (available at `https://www.openblas.net/` and `https://software.intel.com/en-us/mkl`, respectively)
- CMake (available at `https://cmake.org/download/`)

All of those software needs to be installed except for Armadillo which the used can simply download (the include folder is the only required part). Check if those softwares have already been installed by the administrators of your system.

Armadillo [16, 17] handles all algebraic operations. It uses HDF5 to load data, and lapack and OpenBlas (or MKL) for optimization of the mathematical computations. CMake is used to link all libraries for compilations.

The code was tested under Linux OS (Ubuntu 18.04 and CentOS 7), with gcc supporting C++11 and above, Armadillo **v9.100.5**, HDF5 **v1.10.5**, OpenBLAS **v0.2.19**, MKL **v2019.3.199**, Lapack **v3.8.0**, and CMake **v2.8** and above. By default, the software use MKL if installed, otherwise OpenBLAS.

2.2 Building the library

To set up the environment to use the NMFIF package, the user first need to provide the locations for all libraries in the *CMakeList.txt* file. As shown in Listing 1, the user need to provide 3 paths. The paths to provide are indicated in red and correspond to the absolute path to

- Armadillo *include/* folder
- HDF5 *include/* and *lib/* folder

Then open a terminal and navigate to the folder containing the *CMakeList.txt* file and run the following commands to build and compile the library

```
$ cmake .          #build from CMakeList.txt (do not forget the dot)
$ make             #compile library shared object
```

This will create a shared library object .so in the folder. It simply needs to be linked to a project to be able to use our software.

Listing 1: CMakeList.txt file to build library. Blue text correspond to comments and red text indicate which library path to add.

```
cmake_minimum_required(VERSION 2.8)

set(CMAKE_CXX_STANDARD 11)

project(nmfif CXX)

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "cmake_aux/")

include(ARMA_FindMKL)
include(ARMA_FindOpenBLAS)
include(ARMA_FindLAPACK)

if(MKL_FOUND)
  set(LIBS ${LIBS} ${MKL_LIBRARIES})
else()
  if (OpenBLAS_FOUND)
    set(LIBS ${LIBS} ${OpenBLAS_LIBRARIES})
  else()
    message(FATAL_ERROR "MKL or OpenBLAS libraries not found")
  endif()
endif()

if(LAPACK_FOUND)
  set(LIBS ${LIBS} ${LAPACK_LIBRARIES})
else()
  message(FATAL_ERROR "Lapack library not found")
endif()

# Replace PATH_TO_HDF5_LIB_FOLDER
find_library(HDF5_LIB hdf5 PATH_TO_HDF5_LIB_FOLDER)
if(NOT HDF5_LIB)
  message(FATAL_ERROR "hdf5 library not found")
endif()

set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++1y -O3 -march=native -fopenmp -
    DARMA_DONT_USE_WRAPPER -DARMA_USE_HDF5")

add_library(${PROJECT_NAME} SHARED "source/factor.cpp" "source/objective.cpp" "
    source/nmfobjective.cpp" "source/nmtfobjective.cpp" "source/snmtfobjective.cpp"
    "source/integration.cpp")
# Replace PATH_TO_HDF5_INCLUDE_FOLDER and PATH_TO_ARMADILLO_INCLUDE_FOLDER
target_include_directories(${PROJECT_NAME} PUBLIC PATH_TO_HDF5_INCLUDE_FOLDER
    PATH_TO_ARMADILLO_INCLUDE_FOLDER)
target_link_libraries(${PROJECT_NAME} PUBLIC ${HDF5_LIB} ${LIBS})
```

2.3 Linking to a project

Assuming that a *main.cpp* file has been written (see Section 5 for examples of *main.cpp* files), to link the compiled NMFIF library and build the TEST executable simply copy the CMakeList.txt file given in Listing 2 in the folder containing the main.cpp file and run the same two commands and then run the executable

```
$ cmake .          #build from CMakeList.txt (do not forget the dot)
$ make             #compile executable
$ ./TEST           #run executable
```

Listing 2: CMakeList.txt file to compile a project.

```
cmake_minimum_required(VERSION 2.8)
# Replace text highlighted in red by the corresponding absolute path
project(TEST CXX)

set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++1y -fopenmp -DARMA_DONT_USE_WRAPPER
    -DARMA_USE_HDF5")
add_executable(${PROJECT_NAME} "main.cpp")

find_library(NMFIF_LIB nmfif PATH_TO_FOLDER_CONTAINING_NMFIF_SHARED_OBJECT)
if(NOT NMFIF_LIB)
  message(FATAL_ERROR "nmfif library not found")
endif()
find_library(HDF5_LIB hdf5 PATH_TO_HDF5_LIB_FOLDER)
if(NOT HDF5_LIB)
  message(FATAL_ERROR "hdf5 library not found")
endif()

target_link_libraries(${PROJECT_NAME} PUBLIC ${NMFIF_LIB} ${HDF5_LIB})
target_include_directories(${PROJECT_NAME} PUBLIC PATH_TO_NMFIF_SOURCE_FOLDER
    PATH_TO_HDF5_INCLUDE_FOLDER PATH_TO_ARMADILLO_INCLUDE_FOLDER)
```

3 Software details

3.1 Non-negative matrix factorizations

Matrix factorizations approaches aim to approximate a matrix X by the product of n smaller matrices $F_i, i \in \{1..n\}$, called factors, i.e. $X \approx \prod_i F_i$. Mathematically, this amounts to finding factors F_i, under user defined dimensional contraints, that minimize the equation $\|X - \prod_i F_i\|_F$, where $\|\cdot\|_F$ represents the frobenius norm of a matrix and X is the matrix decomposed. Non-negative matrix factorizations techniques add a positivity contraint on the factors, i.e. $\forall i, F_i \geq 0$.

The objective is to obtain lower dimensional representation that capture the essence of the data and can be used to identify missing entries through the matrix completion property. Our software includes four variants of non-negative matrix factorizations approaches.

NMF decomposes a rectangular matrix $X \in \mathbb{R}^{m\times n}$ in the product of two positive factors $F \in \mathbb{R}_+^{m\times k}$ and $G \in \mathbb{R}_+^{n\times k}$, with $k \leq \min(m, n)$, such that $\|X - FG^T\|_F^2$ is minimized.

SNTF decomposes a symmetric matrix $X \in \mathbb{R}^{n\times n}$ in the product of two positive factors $G \in \mathbb{R}_+^{n\times k}$, with $k \leq n$, such that $\|X - GG^T\|_F^2$ is minimized.

NMTF decomposes a rectangular matrix $X \in \mathbb{R}^{m\times n}$ in the product of three positive factors $F \in \mathbb{R}_+^{m\times k_1}$, $S \in \mathbb{R}_+^{k_1\times k_2}$ and $G \in \mathbb{R}_+^{n\times k_2}$, with $k_1, k_2 \leq \min(m, n)$, such that $\|X - FSG^T\|_F^2$ is minimized.

SNMTF decomposes a symmetric matrix $X \in \mathbb{R}^{n\times n}$ in the product of two positive factors $G \in \mathbb{R}_+^{n\times k}$ and $S \in \mathbb{R}_+^{k\times k}$, with $k \leq n$, such that $\|X - GSG^T\|_F^2$ is minimized.

3.2 Initializations

The initialization of the factors of the decompositions is a crucial step of the process. We implement two popular strategies.

3.2.1 Random

The matrices are simply initialized using a uniform distribution in the interval $[0, 1]$.

3.2.2 Truncated singular value decomposition based

The initialization strategy based on the truncated SVD for all factors that has shown better performances than random initialization [1, 14] and has the advantage of giving deterministic solutions. Specifically, consider a factor $G \in \mathbb{R}^{n \times k}$ involved in the decomposition of l data matrices $X_i, i \in \{1..l\}$. Without loss of generality, we assume that G is the right hand side factor in the decomposition, i.e. $\forall i, X_i \approx GF_i$, F_i unspecified. We denote U_i the right hand side term in the SVD decomposition of X_i ($X_i = U_i S_i V_i^T$). We introduce $U_i^+ = \max(U_i, 0)$ and $U_i^- = \max(-U_i, 0)$. We then denote by $\tilde{U}_i \in \mathbb{R}^{n \times k}$ the matrix where each column is defined by

$$\tilde{U}_i(j) = \begin{cases} \sqrt{s_i(j)} U_i^+(j), & \text{if } \|U_i^+(j)\| \geq \|U_i^-(j)\| \\ \sqrt{s_i(j)} U_i^-(j), & \text{otherwise} \end{cases}$$

where $\cdot(j)$ indicates the j^{th} column of the associated matrix and $s_i(j)$ is the j^{th} largest singular value of X_i. Factor G is then initalized as $\frac{1}{l} \sum_{i=1}^{l} \tilde{U}_i + \epsilon$. We initialize the central matrix in NMTF decompositions to the identity matrix.

3.3 Optimization

For the iterative optimization process, our framework use multiplicative update rules [11, 4] that are designed to maintain non-negativity. Under the multiplicative update rules, any entries initialized to zero would stay null. Hence, we add a small $\epsilon = 10^{-9}$ everywhere at initialization to allow all entries to vary.

3.4 Termination

The iterative optimisation can be ran either for a user-defined number epochs or until the relative variation of the objective function between two consecutive epochs is lower than a user-defined threshold δ, i.e. when $\frac{|\mathcal{O}_{t+1} - \mathcal{O}_t|}{\mathcal{O}_t} \leq \delta$ where $\mathcal{O}_t$ corresponds to the value of the objective function at iteration t.

3.5 Graph regularization

Regularization techniques have been successful in many machine learning applications and Non-negative Matrix Factorizations are no exception.

3.6 L2-regularization

L2 regularization is often used as a mean to control the magnitude of the entries of a factor by adding the minimization of the sum of their squares to the objective function. For instance, the standard NMF objective function with L2-regularization applied on factor F would be written as

$$\mathcal{O} = \|X - FG^T\|_F^2 + \sum_{i,j} f_{ij}^2,$$

where f_{ij} denote the entries of factor F.

3.7 Graph regularization

Graph regularization is popular regularization technique used for NMF factorizations[8]. It constrains entities that are connected in a graph to have closer embeddings in the latent space of the factorizations. It is defined using the graph Laplacian. Consider a graph $\mathcal{G}$, with adjacency matrix A where entry $a_{ij} = w_{ij}$, w_{ij} being the weight of the link between entities i and j in G. 0 indicates no link. The Laplacian of $\mathcal{G}$ is defined by $L = D - A$, where D

corresponds to the diagonal matrix where entry $d_{ii} = \sum_j a_{ij}$. Using $\mathcal{G}$ as graph regularization for a factor F gives the additional term $\mathrm{Tr}(F^T LF)$ in the objective function, where $\mathrm{Tr}(\cdot)$ corresponds to the trace operator. For instance, the standard NMF objective function with this graph regularization would be written as

$$\mathcal{O} = \|X - FG^T\|_F^2 + \mathrm{Tr}(F^T LF)$$

4 Package classes

4.1 Factor

The Factor class (files *factor.h* and *factor.cpp*) implements factor centric functions. The crucial functions here are responsible for initializing a factor object and adding regularizers. The prototypes of those methods are given in Listing 3.

Listing 3: Factor method prototypes

```
1  factor(const std::string &output_path, unsigned int m, unsigned int n, const std
       ::string& initMode);
2  void addGraphRegularizer(double* regularizer_adjacency, double lambda);
3  void addL2Regularizer(double lambda);
4  void writeToFile();
```

To initialize a factor, the user need to pass four arguments to the *factor* methods

- A path used to save the factor at the end of optimization by calling the method *writeToFile()*
- The number of rows of the factor
- The number of columns of the factor
- The mode of initialization, one of "random" or "tsvd"

For example, to randomly initialize a factor $F \in \mathbb{R}_+^{50\times 10}$, the user should enter

```
1  factor F(output_path, 50, 10, ''random'');
```

Adding regularizer with scaling hyperparameter lambda is done simply by calling the appropriate method on the factor such as

```
1  F.addGraphRegularizer(pointer_to_regularizer_adjacency,lambda);
2  F.addL2Regularizer(lambda);
```

By calling the *writeToFile()* method, they user can then write the factor to the disk in HDF5 format

```
1  F.writeToFile();
```

4.2 Objective

All the NMF variants are implemented with the objective classes. The class Objective (files *objective.h* and *objective.cpp*) is a virtual class that implements the methods that are common to all NMF variants. Each variant has its own initialization, loss, and differentiation methods. We cover below the different methods available to initialize objectives.

Each objective class has three different initialization methods. The first one used for random initialization or to compute the SVD decomposition. The second one to provide the three matrices of the SVD decompositions directly. And the final method is used to provide paths to precomputed SVD decompositions matrices. The last two methods are useful when performing multiple optimizations procedure on the same data to avoid the overhead of computing the SVD decomposition every time.

4.2.1 nmfObjective

The nmfObjective class (source files nmfobjective.h and nmfobjective.cpp) implements the NMF decomposition. The prototypes to initialize a NMF objective object are given in Listing 4.

Listing 4: nmfOptimization initialization method prototypes

```
1  nmfObjective(double *target, const arma::SizeMat &size, factor*, factor*, bool =
       false, double = 1.0);
2  nmfObjective(double *target, const arma::SizeMat &size, factor*, factor*, arma::
       mat &, arma::mat &, arma::mat &, double = 1.0);
3  nmfObjective(double *target, const arma::SizeMat &size, factor*, factor*, std::
       string, std::string, std::string, double = 1.0);
```

Fiver input arguments are shared by all prototypes: a pointer *target* to a matrix that corresponds to the matrix being decomposed, the *size* of the target matrix, the two factors used in the NMF decomposition, and a float scalar that is used to weigh this objective (default 1.0). The scalar is useful in joint optimization problems to give different weights to different decompositions. The first method accepts a boolean argument. When set to true, the algorithm computed the SVD decomposition of the target matrix. We use the truncated SVD algorithm computing a specific number of eigenvalues. The second and third methods take three additional parameters corresponding to either the three matrices obtained from an SVD decomposition of the target matrix or the paths to those three matrices on the disk.

For instance, a user could initialize the NMF optimization to decompose $X \in \mathbb{R}^{50\times 80}$ in the product FG^T with $F \in \mathbb{R}_+^{50\times 10}$ and $G \in \mathbb{R}_+^{80\times 10}$

```
1  factor F(output_path, 50, 10, ``tsvd'');
2  factor G(output_path, 80, 10, ``tsvd'');
3  nmfObjective O(X.memptr(), arma::size(X), &F, &G, true); // adding argument true
       is necessary to compute the svd decomposition required by the
       initialization method chosen
```

4.2.2 snmfObjective

The snmfObjective class (source files snmfobjective.h and snmfobjective.cpp) implements the standard SNMF decomposition. The prototypes to initialize a SNMF objective object are given in Listing 5.

Listing 5: snmfOptimization initialization method prototypes

```
1  snmfObjective(double *target, const arma::SizeMat &size, factor*, bool = false,
       double = 1.0);
2  snmfObjective(double *target, const arma::SizeMat &size, factor*, arma::mat &,
       arma::mat &, arma::mat &, double = 1.0);
3  snmfObjective(double *target, const arma::SizeMat &size, factor*, std::string,
       std::string, std::string, double = 1.0);
```

The same observations hold for this class, the difference comes only from the presence of a unique factor in the decomposition.

4.2.3 nmtfObjective

The nmtfObjective class (source files nmtfobjective.h and nmtfobjective.cpp) implements the standard NMTF decomposition. The prototypes to initialize a NMTF objective object are given in Listing 6.

Listing 6: nmtfOptimization initialization method prototypes

```
1  nmtfObjective(double *target, const arma::SizeMat &size, factor*, factor*,
       factor*, bool = false, double = 1.0);
2  nmtfObjective(double *target, const arma::SizeMat &size, factor*, factor*, arma
       ::mat &, arma::mat &, arma::mat &, double = 1.0);
3  nmtfObjective(double *target, const arma::SizeMat &size, factor*, factor*,
       factor*, std::string, std::string, std::string, double = 1.0);
```

Here the objective requires three factors for the decomposition.

4.2.4 snmtfObjective

The snmtfObjective class (source files snmtfobjective.h and snmtfobjective.cpp) implements the standard SNMTF decomposition. The prototypes to initialize a SNMTF objective object are given in Listing 7.

Listing 7: snmtfOptimization initialization method prototypes

```
1  nmtfObjective(double *target, const arma::SizeMat &size, factor*, factor*, bool
       = false, double = 1.0);
2  nmtfObjective(double *target, const arma::SizeMat &size, factor*, arma::mat &,
       arma::mat &, arma::mat &, double = 1.0);
3  nmtfObjective(double *target, const arma::SizeMat &size, factor*, factor*, std::
       string, std::string, std::string, double = 1.0);
```

Here the objective only requires two factors.

4.3 Integration

The integration class (*integration.h* and *integration.cpp*) brings all the pieces together to create the joint optimization problem. The class method prototypes are given in Listing 8

Listing 8: Integration class method prototypes

```
1  integration(objectives &o_list, factors &f_list);
2  void optimize(const unsigned int max_iter, double threshold, const std::string &
       filename);
```

To initialize an integration object simply pass to the method a list of objectives and a list of factors. To run the optimization procedure, call the method *optimize* specifying the maximum number of iteration, the threshold for convergence, and a path where to save the loss of all objectives at each iteration. For instance, to run the optimization procedure until convergence reaches 10^{-5} or until the number of iterations reaches 1000, a user should write

```
1 integration I(objectives_list,factors_list); //initialize integration object
2 I.optimize(1000,10e-5,output_path);          //run optimization procedure
```

5 Examples of main.cpp files

We detail below a few examples of *main.cpp* for various optimization problems. To read HDF5 data in C++, matrices in Armadillo have the method *load()* which takes two or three arguments: the path to the file on disk (relative or absolute), the name of the dataset used when saving the HDF5 file (default is 'dataset'), and optionally a transpose operation flag stipulating that the matrix should be transposed when read. This last option might be necessary as different software might save a matrix in different order (column- or row-dominant) and the matrices might need to be transposed when read in.

5.1 Standard NMF decomposition

Consider the standard NMF decomposition of a rectangular matrix $X \in \mathbb{R}^{m \times n}$ in the product of two positive factors $F \in \mathbb{R}_+^{m \times k}$ and $G \in \mathbb{R}_+^{n \times k}$, with $k = 20$, and L2-regularization on F. The problem is thus to minimize the objective $\mathbb{O} = \|X - FG^T\|_F + \sum_{i,j} f_{ij}$. The *main.cpp* file should look like Listing 9.

Listing 9: Standard NMF example

```
1 #include "source/integration.h" //include header file
2
3 int main()
4 {
5     // Check that the matrix read is as you expect, the option arma::hdf5_opts::
          trans perform transposition if it isn't
6     arma::mat X; X.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
          hdf5_opts::trans));
7
```

```
8       unsigned int n1 = arma::size(X)(0);
9       unsigned int n2 = arma::size(X)(1);
10
11      unsigned int k = 20; // set latent space size
12      std::string initializer = ''tsvd''; // choose initializer in {''random'',''tsvd
            ''}
13
14      factors all_factors; //list containing all factors
15      // We first create all the factors and add them to the list of factors
16      factor F("path_for_save",n1,k,initializer); F.addL2Regularizer(1.0);
17      all_factors.push_back(&F);
18      factor G("path_for_save",n2,k,initializer);
19      all_factors.push_back(&G);
20
21      objectives all_objectives; //list containing all objectives
22      // Initialize an NMF objective and add it to the list of objectives
23      nmfObjective O(X.memptr(),arma::size(X),&F,&G,true);
24      all_objectives.push_back(&O);
25
26      // Finally we create an integration object with all factors and objectives
27      integration test(all_objectives, all_factors);
28      test.optimize(max_number_of_iterations, loss_threshold, path_to_output_losses);
29      return 0;
30  }
```

5.2 Joint NMTF decomposition

Now consider the problem tackled by Gligorijević *et al.* which jointly optimizes two NMTF objectives with one shared factor and graph regularizations added to two factors. The joint objective is written as

$$\mathcal{L} = \|X - US_1V^T\|_F^2 + \|Y - WS_2V^T\|_F^2 + \mathrm{Tr}(V^TL_1V) + \mathrm{Tr}(W^TL_2W),$$

which translates into the *main.cpp* given by Listing 10

Listing 10: Joint NMTF example

```
1   #include "source/integration.h" //include header file
2
3   int main()
4   {
5       // Check that the matrix read is as you expect, the option arma::hdf5_opts::
            trans perform transposition if it isn't
6       arma::mat X; X.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
            hdf5_opts::trans));
7       arma::mat Y; Y.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
            hdf5_opts::trans));
8       arma::mat A1; A1.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma
            ::hdf5_opts::trans));
9       arma::mat A2; A2.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma
            ::hdf5_opts::trans));
10
11      unsigned int n1 = arma::size(X)(0);
12      unsigned int n2 = arma::size(X)(1);
13      unsigned int n3 = arma::size(Y)(0);
14
15      unsigned int k1; unsigned int k2=100; unsigned int k3=100; // set latent spaces
            size
16      std::string initializer = ''random''; // choose initializer in {''random'',''
            tsvd''}
```

```
17
18      factors all_factors; //initialize empty list to store factors
19      // We first create all the factors and add them to the list of factors
20      factor U("path_for_save",n1,k1,initializer);  all_factors.push_back(&U);
21      factor S1("path_for_save",k1,k2,initializer); all_factors.push_back(&S1);
22      factor V("path_for_save",n2,k2,initializer);
23      V.addGraphRegularizer(&A1,1.0); all_factors.push_back(&V); // adding graph
            regularization before adding to list
24      factor S2("path_for_save",k3,k2,initializer); all_factors.push_back(&S2);
25      factor W("path_for_save",n3,k3,initializer);
26      W.addGraphRegularizer(&A2,1.0); all_factors.push_back(&W);
27
28      objectives all_objectives; //initialize empty list to store objectives
29      // Initialize  NMF objectives and add to the list of objectives
30      nmtfObjective O1(X.memptr(),arma::size(X),&U,&S1,&V,false); // adding false is
            not necessary as it is the default
31      nmtfObjective O2(Y.memptr(),arma::size(Y),&W,&S2,&V);
32      all_objectives.push_back(&O1); all_objectives.push_back(&O2);
33
34      // Finally we create an integration object with all factors and objectives
35      integration test(all_objectives, all_factors);
36      test.optimize(max_number_of_iterations, loss_threshold, path_to_output_losses);
37      return 0;
38  }
```

5.3 Joint mixed NMF problem

Our software can handle any problem with mixed variants of NMF. For instance consider the following objective function

$$\mathcal{O} = 0.5\|X - UV^T\|_F^2 + 0.3\|Y - WW^T\|_F^2 + 0.2\|Z - USW^T\|_F^2 + 0.01\sum_{i,j} v_{ij},$$

where $X \in \mathbb{R}^{1000\times5000}$, $Y \in \mathbb{R}^{69420\times69420}$, $Z \in \mathbb{R}^{1000\times69420}$, $U \in \mathbb{R}^{1000\times50}$, $V \in \mathbb{R}^{5000\times50}$, $W \in \mathbb{R}^{69420\times333}$, and $S \in \mathbb{R}^{50\times333}$.

Listing 11: Mixed NMF example

```
1   #include "source/integration.h" //include header file
2
3   int main()
4   {
5       // Check that the matrix read is as you expect, the option arma::hdf5_opts::
            trans perform transposition if it isn't
6       arma::mat X; X.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
            hdf5_opts::trans));
7       arma::mat Y; Y.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
            hdf5_opts::trans));
8       arma::mat Z; Z.load(arma::hdf5_name("path_to_hdf5_format_file","dataset",arma::
            hdf5_opts::trans));
9
10      unsigned int n1 = 1000; unsigned int n2 = 5000; unsigned int n3 = 69420;
11
12      unsigned int k1 = 50; unsigned int k2=333; // set latent spaces size
13      std::string initializer = ''tsvd''; // choose initializer in {''random'',''tsvd
            ''}
14
15      factors all_factors; //initialize empty list to store factors
```

```
16      // We first create all the factors and add them to the list of factors
17      factor U("path_for_save",n1,k1,initializer);   all_factors.push_back(&U);
18      factor V("path_for_save",n2,k1,initializer); V.addL2Regularizer(0.01);
            all_factors.push_back(&V);
19      factor W("path_for_save",n3,k2,initializer);   all_factors.push_back(&W);
20      factor S("path_for_save",k1,k2,initializer); all_factors.push_back(&S);
21
22      objectives all_objectives; //initialize empty list to store objectives
23      // Initialize  NMF objectives and add to the list of objectives
24      nmfObjective O1(X.memptr(),arma::size(X),&U,&S1,&V,true,0.5);
25      snmfObjective O2(Y.memptr(),arma::size(Y),&W,true,0.1);
26      nmtfObjective O3(Z.memptr(),arma::size(Y),&U,&S,&W,true,0.2);
27      all_objectives.push_back(&O1); all_objectives.push_back(&O2);
28      all_objectives.push_back(&O3);
29
30      // Finally we create an integration object with all factors and objectives
31      integration test(all_objectives, all_factors);
32      test.optimize(max_number_of_iterations, loss_threshold, path_to_output_losses);
33      return 0;
34  }
```

References

[1] Christos Boutsidis and Efstratios Gallopoulos. Svd based initialization: A head start for nonnegative matrix factorization. *Pattern Recognition*, 41(4):1350–1362, 2008.

[2] Prabhakar Chalise and Brooke L Fridley. Integrative clustering of multi-level 'omic data based on non-negative matrix factorization algorithm. *PloS One*, 12(5), 2017.

[3] Andrew Chatr-Aryamontri, Rose Oughtred, Lorrie Boucher, Jennifer Rust, Christie Chang, Nadine K Kolas, Lara O'Donnell, Sara Oster, Chandra Theesfeld, Adnane Sellam, et al. The biogrid interaction database: 2017 update. *Nucleic Acids Research*, 45(D1):D369–D379, 2017.

[4] Andrej Čopar, Blaž Zupan, et al. Scalable non-negative matrix tri-factorization. *BioData Mining*, 10(1):41, 2017.

[5] Chris Ding, Tao Li, Wei Peng, and Haesun Park. Orthogonal nonnegative matrix t-factorizations for clustering. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 126–135, 2006.

[6] Ron Edgar, Michael Domrachev, and Alex E Lash. Gene expression omnibus: Ncbi gene expression and hybridization array data repository. *Nucleic acids research*, 30(1):207–210, 2002.

[7] Antonio Fabregat, Steven Jupe, Lisa Matthews, Konstantinos Sidiropoulos, Marc Gillespie, Phani Garapati, Robin Haw, Bijay Jassal, Florian Korninger, Bruce May, Marija Milacic, Corina Duenas Roca, Karen Rothfels, Cristoffer Sevilla, Veronica Shamovsky, Solomon Shorser, Thawfeek Varusai, Guilherme Viteri, Joel Weiser, Guanming Wu, Lincoln Stein, Henning Hermjakob, and Peter D'Eustachio. The reactome pathway knowledgebase. *Nucleic Acids Research*, 46(D1):D649–D655, 2018.

[8] Vladimir Gligorijević, Noël Malod-Dognin, and Nataša Pržulj. Patient-specific data fusion for cancer stratification and personalised treatment. In *Biocomputing 2016: Proceedings of the Pacific Symposium*, pages 321–332. World Scientific, 2016.

[9] Matan Hofree, John P Shen, et al. Network-based stratification of tumor mutations. *Nature Methods*, 10(11):1108–1115, 2013.

[10] Sijia Huang, Kumardeep Chaudhary, and Lana X Garmire. More is better: recent progress in multi-omics data integration methods. *Frontiers in Genetics*, 8:84, 2017.

[11] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[12] Mark DM Leiserson, Fabio Vandin, et al. Pan-cancer network analysis identifies combinations of rare somatic mutations across pathways and protein complexes. *Nature Genetics*, 47(2):106, 2015.

[13] Bo Long, Zhongfei Mark Zhang, and Philip S Yu. Co-clustering by block value decomposition. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, pages 635–640. ACM, 2005.

[14] Noël Malod-Dognin, Julia Petschnigg, et al. Towards a data-integrated cell. *Nature Communications*, 10(1):1–13, 2019.

[15] Rachana Mehta and Keyur Rana. A review on matrix factorization techniques in recommender systems. In *2017 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA)*, pages 269–274. IEEE, 2017.

[16] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016.

[17] Conrad Sanderson and Ryan Curtin. A user-friendly hybrid sparse matrix class in c++. In *International Congress on Mathematical Software*, pages 422–430. Springer, 2018.

[18] Yan V Sun and Yi-Juan Hu. Integrative analysis of multi-omics data for discovery and functional studies of complex human diseases. In *Advances in Genetics*, volume 93, pages 147–190. Elsevier, 2016.

[19] Prashanth Suravajhala, Lisette JA Kogelman, and Haja N Kadarmideen. Multi-omic data integration and analysis using systems genomics approaches: methods and applications in animal production, health and welfare. *Genetics Selection Evolution*, 48(1):38, 2016.

[20] Yu-Xiong Wang and Yu-Jin Zhang. Nonnegative matrix factorization: A comprehensive review. *IEEE Transactions on Knowledge and Data Engineering*, 25(6):1336–1353, 2012.

[21] Zi Yang and George Michailidis. A non-negative matrix factorization method for detecting modules in heterogeneous omics multi-modal data. *Bioinformatics*, 32(1):1–8, 2016.

[22] Junjun Zhang, Rosita Bajari, Dusan Andric, Francois Gerthoffert, Alexandru Lepsa, Hardeep Nahal-Bose, Lincoln D Stein, and Vincent Ferretti. The international cancer genome consortium data portal. *Nature biotechnology*, 37(4):367–369, 2019.

[23] Zhang Zhang, Vladimir B Bajic, et al. Data integration in bioinformatics: current efforts and challenges. *Bioinformatics-Trends and Methodologies*, pages 41–56, 2011.