SmallK: A Library for Nonnegative Matrix Factorization, Topic Modeling, and Clustering of Large-Scale Data

Prepared by
Richard Boyd[1,2], Barry Drake[1,2], Da Kuang[2] and Haesun Park[2]

Point of Contact:
Professor Haesun Park
hpark@cc.gatech.edu

[1]Cyber Technology and Information Security Laboratory
Georgia Tech Research Institute
250 14th St. NW
Atlanta, GA 30318

[2]School of Computational Science and Engineering
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

8 August 2014

I. Introduction

High-dimensional data sets are ubiquitous in data science, and they often present serious problems for researchers. These data sets cannot be directly visualized, and most of the available algorithms used for the analysis and classification of large data sets suffer from the "curse of dimensionality", which sharply reduces the speed and accuracy of these algorithms. In reality, most high-dimensional data sets fill only a subspace of the whole space of possible values. For instance, there are common correlations between the results of various medical diagnostic measurements, and certain values are not likely to occur together. Thus, while the full data space may be high dimensional, the data itself usually occupy regions of much lower intrinsic dimensionality.

Our work in dimensionality reduction focuses on, but is not limited to, low-rank approximations via nonnegative matrix factorization (NMF) [1, 2]. NMF is a non-convex optimization problem with important applications in data and interactive visual analytics of high-dimensional data.

The impetus for this document is to provide a step-by-step procedure for the application of the theory to real-world large-scale data and visual analytics. We have instantiated our research efforts in a software framework that includes high-level driver code via Python and a simple command line interface, SmallK, which hides most of the details of the input parameters. Our low-level code, also usable from the command line, is written in C++, which provides efficient NMF algorithms. The algorithms discussed herein have numerous practical applications; this document will hopefully provide the information required to quickly begin real work.

Below is a brief description of our fundamental research on NMF algorithms (please see the References section for more detail). Following the brief motivational introduction to the NMF are detailed installation instructions for the NMF software framework.

I.1 Low-rank approximations and NMF

Algorithms that enable dimension reduction and clustering are two critical areas in data analytics and interactive visual analysis of high-dimensional data. A low-rank approximation framework has the ability to facilitate faster processing times and utilize fewer resources. These approximations provide a natural way to compute only what we need for significant dimension reduction, and are analogous to singular value decomposition (SVD) and principal component analysis (PCA). The algorithm framework also works efficiently for clustering since clustering can be viewed as a specific way of achieving a low-rank approximation so that the clustered structure of the data is well represented in a few basis vectors.

Matrix factorizations such as the SVD have played a key role as a fundamental tool in machine learning, data mining, and other areas of computational science and engineering. The NMF has recently emerged as an important factorization method as well. A distinguishing feature of the NMF is the requirement of nonnegativity: NMF is considered for high-dimensional and large scale data in which the representation of each element is inherently nonnegative, and it seeks low-rank factor matrices that are constrained to have only nonnegative elements. There are many examples of data with a nonnegative representation. In a standard term-frequency encoding, a text document is represented as a vector of nonnegative numbers since each element represents the number of appearances of each term in each document. In image processing, digital images are represented by pixel intensities, which are nonnegative. In the life sciences, chemical concentrations or gene expression levels are naturally represented as nonnegative data.

Our algorithm framework utilizes various constraints on the non-convex optimization problem that gives rise to the nonnegative factors. With these various constraints NMF is a versatile tool for a large variety of data analytics problems. NMF algorithms have been an active area of research for several years. Since much of the data for many important problems in numerous domains is nonnegative NMF is the correct computational model for mining and/or integrating information from such data. NMF also offers enhanced interpretation of results since nonnegativity of the data is preserved.

II. SmallK Overview

The SmallK library provides routines for low-rank matrix approximation via nonnegative matrix factorization (NMF). The term “nonnegative matrices” means that for a given matrix Z all elements of Z are greater than or

equal to zero, which we express as $\geq \mathbf{0}$. Given a nonnegative matrix A, the SmallK software computes nonnegative matrices W and H.

$$\mathbf{A} \cong \mathbf{W}\,\mathbf{H}$$

The matrix A has m rows and n columns and can be either sparse or dense. W has m rows and k columns, and H has k rows and n columns. W and H are always dense, even when A is sparse. The value of k is an input parameter to the approximation routines; typically k << m and k << n. k is expressed mathematically as the rank of the low rank approximation.

NMF algorithms seek to approximate a matrix A by the product of two much smaller matrices W and H. The idea is to choose the smallest value of k (width of W and height of H) that gives an acceptable approximation error. Due to the nonconvex nature of the optimization problem associated with finding W and H, they can only be approximated after an NMF algorithm satisfies a convergence criterion. Thus, the minimization of the objective function proceeds iteratively, attempting to reach a stationary point, which is the best possible solution. As the iterations proceed, the SmallK code computes a metric that estimates the progress and, when the metric falls below a user-specified tolerance, the iterations stop and convergence is declared.

The SmallK library provides implementations of several NMF algorithms. These algorithms are:

1. Multiplicative Updating (NMF-MU)
2. Hierarchical Alternating Least Squares (NMF-HALS)
3. Block Principal Pivoting (NMF-BPP)
4. Rank2 Specialization (NMF-RANK2)

Additional NMF algorithms will be provided in future updates.

SmallK also provides implementations of hierarchical and flat data clustering. These routines are:

1. Hierarchical Clustering via NMF-RANK2
2. Flat Clustering via NMF-RANK2
3. Flat Clustering via NMF-BPP or NMF-HALS

The suite of SmallK implementations of NMF algorithms are suitable in many applications such as image processing, interactive visual analytics, speckle removal from SAR images, recommender systems, information fusion, outlier detection, chemometrics, and many more.

The SmallK library requires either MacOSX or Linux. A Windows version may be provided in the future.

III. Build and Installation Instructions

III.1 Quick Start: Build a SmallK Virtual Machine using Vagrant

Installing SmallK into a virtual machine is intended for those who are not doing development and/or do not have a reason to do the full installation on Linux or OSX outlined in sections III.2 to III.4.

The complete stack of software dependencies for SmallK as well as SmallK itself can be rapidly set up and configured through use of Vagrant and VirtualBox and the files included in the repository. To deploy the SmallK VM:

1. Install Vagrant and VirtualBox.

2. From within the vagrant/ directory in the repository run:

```
vagrant up
```

This can take as long as an hour to build the VM, which will be based on a minimal Ubuntu 14.04 installation. The VagrantFile can be customized in many ways to change the specifications for the VM that is built. See more information here.

3. Once the VM has been built, run:

```
vagrant ssh
```

This will drop you into the command line of the VM that was just created. From there, you can navigate to /home/vagrant/smallk-1.4.0 and run

```
make check
```

to verify your installation was successful. In case you need it, the username/password for the VM created will be vagrant/vagrant.

4. When you are ready to shut down the VM, run one of the following:

```
vagrant suspend # this command will save the current running state

vagrant halt # this command will gracefully shut down the machine

vagrant destroy #this command will remove the VM from your machine
```

If you want to work with the VM again, from any of the above states you can run

```
vagrant up
```

again and the VM will be resumed or recreated.

III.2 Standard Build and Installation

Prerequisites

- A modern C++ compiler that supports the C++11 standard, such as the latest release of the GNU or clang compilers
- Elemental, a high-performance library for dense, distributed linear algebra, which requires:
  - An MPI installation, such as OpenMPI
  - A BLAS implementation, preferably optimized/tuned for the local system
  - libFLAME: a high-performance library for dense linear algebra
  - OpenMP: (optional)
  - CMake

Elemental can make use of OpenMP parallelization if available. This is generally advantageous for large problems. The SmallK code is also internally parallelized to take full advantage of multiple CPU cores for maximum performance. SmallK does not currently support distributed computation. However, future updates are planned that provide this capability.

The SmallK software supports the latest stable release of Elemental, version 0.84.

A note of caution: copying the command lines from this document and pasting them into a terminal may result in the commands not properly executing due to how Word interprets the double dash --, "double quotes", and perhaps other characters or symbols. For pasting the commands to a terminal, first copy the command lines to a text editor and copy/paste from there.

III.3 How to Install Elemental on MacOSX

On MacOSX we recommend using Homebrew as the package manager. Homebrew does not require sudo privileges for package installation, unlike other package managers such as MacPorts. Thus the chances of corrupting vital system files are greatly reduced with Homebrew.

It is convenient to be able to view hidden files (like .file) in the MacOSX Finder. To do so run the following at the command line:

```
defaults write com.apple.finder AppleShowAllFiles -bool YES
```

To hide hidden files again, set the Boolean flag to NO:

```
defaults write com.apple.finder AppleShowAllFiles -bool NO
```

If you use Homebrew, ensure that your PATH is configured to search Homebrew's installation directory first. Homebrew's default installation location is /usr/local/bin, so that location needs to be first on your path. To check, run this command from a terminal window:

```
cat /etc/paths
```

If the first entry is not /usr/local/bin, you will need to edit this file. Since this is a system file, first create a backup. If permission is denied to modify this file, use `sudo` in front of the commands. Move the line `/usr/local/bin` so that it is on the first line of the file. Save the file, then close the terminal session and start a new terminal session so that the path changes will take effect.

III.3.1 OSX:Install the latest GNU and clang compilers

Elemental and SmallK both require a modern C++ compiler compliant with the C++11 standard. We recommend that you install the latest stable version of the clang and GNU C++ compilers. To do this, first install the XCode command line tools with this command:

```
xcode-select --install
```

If this command produces an error, download and install XCode from the AppStore, then repeat the command. If that should still fail, install the command line tools from the XCode preferences menu. After the installation completes, run this command from a terminal window to check the version of the clang compiler:

```
clang++ --version
```

You should see output similar to this:

```
Apple LLVM version 5.1 (clang-503.0.40) (based on LLVM 3.4svn)
Target: x86-64-apple-darwin13.2.0
Thread model: posix
```

The latest version of the GNU compiler at the time of writing is g++-4.9. This can be installed with Homebrew as follows:

```
brew tap homebrew/versions
brew install gcc49 --enable-fortran
```

The Apple-provided gcc and g++ will not be overwritten by this installation. The new compilers will be installed into /usr/local/bin as gcc-4.9, g++-4.9, and gfortran-4.9. The Fortran compiler is needed for the installation of MPI.

III.3.2 OSX:Install OpenMPI

Install the latest version of OpenMPI with Homebrew as follows:

```
brew install open-mpi --c++11
```

The Homebrew install formula provides an option for "thread-multiple" support, but do not enable this option, as it is still experimental, not optimized for performance, and may have bugs.

III.3.3 OSX:Install the latest version of libFLAME

Next we detail the installation of the high performance numerical library libflame. The library can be gotten from the libflame git repository on github.

It's important to perform the git clone into a subdirectory NOT called 'flame' since this can cause name conflicts with the installation. We normally do a git clone into a directory called 'libflame'. However, other directory names

will work as well, but not 'flame'.

To obtain the latest version of the FLAME library, clone the FLAME git repository with this command:

```
git clone https://github.com/flame/libflame.git <libflame>
```

Run the configure script in the top-level FLAME folder as follows (assuming you want to install to /usr/local/flame; if not, change the prefix path):

```
./configure --prefix=/usr/local/flame --with-cc=/usr/local/bin/gcc-4.9 --with-
ranlib=/usr/local/bin/gcc-ranlib-4.9
```

A complete list of configuration options can be obtained by running: `./configure --help`.

After the configuration process completes, build the FLAME library as follows:

```
make -j4
```

The `-j4` option tells Make to use four processes to perform the build. This number can be increased if you have a more capable system.

```
make install
```

The FLAME library is now installed.

III.3.4 OSX:Install Elemental

We **strongly** recommend that users install both the HybridRelease and PureRelease builds of Elemental. OpenMP is enabled in the HybridRelease build and disabled in the PureRelease build. So why install both? For smaller problems the overhead of *OpenMP can actually cause code to run slower* than without it. Whereas for large problems OpenMP parallelization generally helps, but there is no clear transition point between where it helps and where it hurts. Thus we encourage users to experiment with both builds to find the one that performs best for their typical problems.

We also recommend that users clearly separate the different build types as well as the versions of Elemental on their systems. Elemental is under active development, and new releases can introduce changes to the API that are not backwards compatible with previous releases. To minimize build problems and overall hassle, we recommend that Elemental be installed so that the different versions and build types are cleanly separated.

Here is our recommended installation scheme for Elemental:

Choose a folder for the root of the Elemental installation. For our systems, this is

```
/usr/local/elemental
```

Inside of this folder create a new folder named with the release version of Elemental:

```
/usr/local/elemental/0.84/
```

Inside of this version folder, create two additional folders for each Elemental build type. These should be named HybridRelase and PureRelease, to match Elemental's terminology. Thus the final folder configuration is

```
/usr/local/elemental/0.84/HybridRelease
/usr/local/elemental/0.84/PureRelease
```

Download the 0.84 release of Elemental, unzip and untar the distribution, and cd to the top-level folder. Run the following command to create a local build folder for the HybridRelease build:

```
mkdir build_hybrid
cd build_hybrid
```

Use the following CMake command for the HybridRelease build:

```
cmake -D CMAKE_INSTALL_PREFIX=/usr/local/elemental/0.84/HybridRelease
-D CMAKE_BUILD_TYPE=HybridRelease
-D CMAKE_CXX_COMPILER=/usr/local/bin/g++-4.9
-D CMAKE_C_COMPILER=/usr/local/bin/gcc-4.9
-D CMAKE_Fortran_COMPILER=/usr/local/bin/gfortran-4.9
-D MATH_LIBS="/usr/local/flame/lib/libflame.a;-framework Accelerate"
-D ELEM_EXAMPLES=ON -D ELEM_TESTS=ON  ..
```

Note that we have installed g++-4.9 into /usr/local/bin and libFLAME into /usr/local/flame. Alter these paths, if necessary, to match the installation location on your system.

Once the CMake configuration step completes, you can build Elemental from the generated Makefiles with the following command:

```
make -j4
```

The `-j4` option tells Make to use four processes to perform the build. This number can be increased if you have a more capable system.

After the build completes, install elemental as follows:

```
make install
```

As a final step, edit the file `/usr/local/elemental/0.84/HybridRelease/conf/ElemVars` and replace the line

```
CXX = /usr/local/bin/g++-4.9
```

With this:

```
CXX = /usr/local/bin/g++-4.9 -std=c++11
```

This will eliminate some compiler warnings about C++11 constructs.

After this, run these commands to create a build folder for the PureRelease build:

```
cd ..
mkdir build_pure
cd build_pure
```

Then repeat the CMake configuration process, this time with the following command for the PureRelease build:

```
cmake -D CMAKE_INSTALL_PREFIX=/usr/local/elemental/0.84/PureRelease
-D CMAKE_BUILD_TYPE=PureRelease -D CMAKE_CXX_COMPILER=/usr/local/bin/g++-4.9
-D CMAKE_C_COMPILER=/usr/local/bin/gcc-4.9
-D CMAKE_Fortran_COMPILER=/usr/local/bin/gfortran-4.9
-D MATH_LIBS="/usr/local/flame/lib/libflame.a;-framework Accelerate"
-D ELEM_EXAMPLES=ON -D ELEM_TESTS=ON  ..
```

Repeat the build commands and install this build of Elemental. Then edit the `/usr/local/elemental/0.84/PureRelease/conf/ElemVars` file and replace the CXX line as indicated above.

This completes the two builds of Elemental.

To test the installation, follow Elemental's test instructions for the SVD test to verify that Elemental is working correctly.

III.4 How to Install Elemental on Linux

We strongly recommend using a package manager for your Linux distribution for installation and configuration of the required dependencies. We cannot provide specific installation commands for every variant of Linux, so we specify the high-level steps below.

III.4.1 Linux:Install the latest GNU compilers

We recommend installation of the latest stable release of the GNU C++ compiler, which is g++-4.9 at the time of this writing.

Also install the latest version of GNU Fortran, which is needed for the installation of MPI.

III.4.2 Linux:Install OpenMPI

Download the latest version of OpenMPI, unzip and untar the downloaded zip file, and cd to the untarred directory. Run configure as follows, all on a single line. This command assumes that gcc-4.9 has been installed; change the paths if needed to match your system:

```
./configure --prefix=/usr/local  CC=/usr/local/bin/gcc-4.9 CXX=/usr/local/bin/g++-4.9
F77=/usr/local/bin/gfortran-4.9  FC=/usr/local/bin/gfortran-4.9
```

Wait for the configure script to finish – this could take several minutes. Then build the code as follows:

```
make -j4
```

Install with:

```
make install
```

OpenMPI provides many tests to verify the installation; it's a good idea to run at least some of these tests to ensure that MPI was installed successfully.

III.4.3 Linux:Install the latest version of libFlame

To obtain the latest version of the FLAME library, clone the FLAME git repository with this command:

```
git clone https://github.com/flame/libflame.git
```

Run the configure script in the top-level FLAME folder as follows (assuming you want to install to /usr/local/flame; if not, change the prefix path):

```
./configure --prefix=/usr/local/flame --with-cc=/usr/local/bin/gcc-4.9 --with-
ranlib=/usr/local/bin/gcc-ranlib-4.9
```

A complete list of configuration options can be obtained by running `./configure --help`.

After the configuration process completes, build the FLAME library as follows:

```
make -j4
make install
```

This completes the installation of the FLAME library.

III.4.4 Linux:Install an accelerated BLAS library

It is essential to link Elemental with an accelerated BLAS library for maximum performance. Linking Elemental with a 'reference' BLAS implementation will cripple performance, since the reference implementations are designed for correctness not speed.

If you do not have an accelerated BLAS on your system, you can download and build OpenBLAS. Download, unzip, and untar the tarball (version 0.2.8 as of this writing) and cd into the top-level folder. Build OpenBLAS with this command, assuming you have a 64-bit system:

```
make BINARY=64 USE_OPENMP=1
```

Install with this command, assuming the installation directory is `/usr/local/openblas/0.2.8/`:

```
make PREFIX=/usr/local/openblas/0.2.8/ install
```

This completes the installation of OpenBLAS.

III.4.5 Linux:Install Elemental

We **strongly** recommend that users install both the HybridRelease and PureRelease builds of Elemental. OpenMP is enabled in the HybridRelease build and disabled in the PureRelease build. So why install both? For smaller problems the overhead of OpenMP can actually cause code to run slower than without it. On the other hand, for large problems, OpenMP parallelization generally helps. However, there is no clear transition point between where it helps and where it hurts. Thus, we encourage users to experiment with both builds to find the one that performs best for their typical problems.

We also recommend that users clearly separate the different build types as well as the versions of Elemental on their systems. Elemental is under active development, and new releases can introduce changes to the API that are not backwards compatible with previous releases. To minimize build problems and overall hassle, we recommend that Elemental be installed so that the different versions and build types are cleanly separated.

Here is our recommended installation scheme for Elemental:

Choose a folder for the root of the Elemental installation. For our systems, this is

```
/usr/local/elemental
```

Inside of this folder create a new folder named with the release version of Elemental:

```
/usr/local/elemental/0.84/
```

Inside of this version folder, create two additional folders for each Elemental build type. These should be named HybridRelase and PureRelease, to match Elemental's terminology. Thus the final folder configuration is

```
/usr/local/elemental/0.84/HybridRelease
/usr/local/elemental/0.84/PureRelease
```

HybridRelease Build

Download the 0.84 release of Elemental, unzip and untar the distribution, and cd to the top-level folder. Run the following command to create a local build folder for the HybridRelease build:

```
mkdir build_hybrid
cd build_hybrid
```

Use the following CMake command for the HybridRelease build:

```
cmake -D CMAKE_INSTALL_PREFIX=/usr/local/elemental/0.84/HybridRelease
-D CMAKE_BUILD_TYPE=HybridRelease -D CMAKE_CXX_COMPILER=/usr/local/bin/g++-4.9
-D CMAKE_C_COMPILER=/usr/local/bin/gcc-4.9
-D CMAKE_Fortran_COMPILER=/usr/local/bin/gfortran-4.9
-D MATH_LIBS="/usr/local/flame/lib/libflame.a;-L/usr/local/openblas/0.2.8/ -lopenblas -
lm"
-D ELEM_EXAMPLES=ON -D ELEM_TESTS=ON  ..
```

Note that we have installed g++-4.9 into /usr/local/bin and libFLAME into /usr/local/flame. Alter these paths, if necessary, to match the installation location on your system.

If this command does not work on your system, you may need to define the BLAS_LIBS and/or GFORTRAN_LIB config options.

Once the CMake configuration step completes, you can build Elemental from the generated Makefiles with the following command:

```
make -j4
```

The `-j4` option tells Make to use four processes to perform the build. This number can be increased if you have a more capable system.

After the build completes, install elemental as follows:

```
make install
```

As a final step, edit the file `/usr/local/elemental/0.84/HybridRelease/conf/ElemVars` and replace the line

```
CXX = /usr/local/bin/g++-4.9
```

With this:

```
CXX = /usr/local/bin/g++-4.9 -std=c++11
```

This will eliminate some compiler warnings about C++11 constructs.

PureRelease Build

After this, run these commands to create a build folder for the PureRelease build:

```
cd ..
mkdir build_pure
cd build_pure
```

Then repeat the CMake configuration process, this time with the following command for the PureRelease build:

```
cmake -D CMAKE_INSTALL_PREFIX=/usr/local/elemental/0.84/PureRelease
-D CMAKE_BUILD_TYPE=PureRelease -D CMAKE_CXX_COMPILER=/usr/local/bin/g++-4.9
-D CMAKE_C_COMPILER=/usr/local/bin/gcc-4.9
-D CMAKE_Fortran_COMPILER=/usr/local/bin/gfortran-4.9
-D MATH_LIBS="/usr/local/flame/lib/libflame.a;-L/usr/local/openblas/0.2.8/ -lopenblas -
lm"
-D ELEM_EXAMPLES=ON -D ELEM_TESTS=ON  ..
```

If this command does not work on your system, you may need to define the BLAS_LIBS and/or GFORTRAN_LIB config options.

Repeat the build commands and install this build of Elemental. Then edit the `/usr/local/elemental/0.84/PureRelease/conf/ElemVars` file and replace the CXX line as indicated above.

This completes the two builds of Elemental.

To test the installation, follow Elemental's test instructions for the SVD test to verify that Elemental is working correctly.

IV. Building and Installing the SmallK Source Code

IV.1 Obtain the source code

The source code for the SmallK library can be obtained by downloading from the [SmallK](https://github.com/smallk/smallk.github.io/tree/master/code) repository on github.

Once downloaded uncompress and follow the installation instructions below.

IV.2 Build the SmallK library

Download and unpack the SmallK code tarball and cd into the top-level SmallK folder. The makefiles assume that you followed our suggested installation plan for Elemental. If this is not the case you will need to do one of the following things:

1. Create an environment variable called ELEMENTAL_INSTALL_DIR which contains the path to the root folder of your Elemental installation
2. Define the variable ELEMENTAL_INSTALL_DIR on the make command line
3. Edit the SmallK makefile so that it can find your Elemental installation

Assuming that the default install locations are acceptable, build the SmallK code by running this command from the root folder of the distribution:

```
make all
```

This will build the SmallK library and several command-line applications. These are:

1. libsmallk.a, the SmallK library
2. preprocess_tf, a command-line application for processing and scoring term-frequency matrices
3. matrixgen, a command-line application for generating random matrices
4. nmf, a command-line application for NMF
5. hierclust, a command-line application for fast hierarchical clustering
6. flatclust, a command-line application for flat clustering via NMF

To install the code, run this command to install to the default location, which is `/usr/local/smallk`:

```
make install
```

This will install the files listed above into the `/usr/local/smallk/bin` directory, which needs to be on your path to run the executables from anywhere and avoid prepending with the entire path. To install the code to a different location, either create an environment variable called SMALLK_INSTALL_DIR and set it equal to the desired installation location prior to running the install command, or supply a prefix argument:

```
make prefix=/path/to/smallk  install
```

Or, as a last resort, you can edit the top-level SmallK makefile to conform to the installation scheme of your system. You may need root privileges to do the installation, depending on where you choose to install it.

To test the installation, run this command:

```
make check
```

This will run a series of tests, none of which should report a failure. Sample output from a run of these tests can be found in section [V.6. SmallK Test Results].

The command-line applications can be built individually by running the appropriate make command from the top-level smallk folder. These commands are:

To build the smallk library only:	`make libsmallk`
To build the preprocessor only:	`make preprocessor`
To build the matrix generator only:	`make matrixgen`
To build the nmf only:	`make nmf`
To build hierclust only:	`make hierclust`
To build flatclust only:	`make flatclust`

IV.3 Examples of API Usage

In the `examples` folder you will find a file called `smallk_example.cpp`. This file contains several examples of how to use the SmallK library. Also included in the examples folder is a makefile that you can customize for your use. Note that the SmallK library must first be installed before the example project can be built.

As an example of how to use the sample project, assume the smallk software has been installed into /usr/local/smallk. Also assume that the user chose to create the recommended environment variable SMALLK_INSTALL_DIR that stores the location of the top-level install folder, i.e. the user's .bashrc file contains this statement:

```
export SMALLK_INSTALL_DIR=/usr/local/smallk
```

To build the smallk example project, open a terminal window and cd to the smallk/examples folder and run this command:

```
make
```

To run the example project, run this command:

```
./bin/example ../data
```

The output will be similar to the following (it won't be identical because some problems are randomly initialized):

```
Smallk major version: 1
Smallk minor version: 0
Smallk patch level:   0
Smallk version string: 1.0.0
Loading matrix...

*************************************************************
*                                                           *
*                Running NMF-BPP using k=32                 *
*                                                           *
*************************************************************
Initializing matrix W...
Initializing matrix H...

                parameters:

               algorithm: Nonnegative Least Squares with Block Principal Pivoting
      stopping criterion: Ratio of Projected Gradients
                  height: 12411
                   width: 7984
                       k: 32
                miniter: 5
                maxiter: 5000
                    tol: 0.005
              matrixfile: ../data/reuters.mtx
              maxthreads: 8

1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.0747031
7:    progress metric:  0.0597987
8:    progress metric:  0.0462878
9:    progress metric:  0.0362883
10:   progress metric:  0.030665
11:   progress metric:  0.0281802
12:   progress metric:  0.0267987
13:   progress metric:  0.0236731
14:   progress metric:  0.0220778
15:   progress metric:  0.0227083
16:   progress metric:  0.0244029
17:   progress metric:  0.0247552
18:   progress metric:  0.0220007
19:   progress metric:  0.0173831
20:   progress metric:  0.0137033

Solution converged after 39 iterations.

Elapsed wall clock time: 4.354 sec.

Writing output files...

*************************************************************
*                                                           *
*                Running NMF-HALS using k=16                *
*                                                           *
*************************************************************
Initializing matrix W...
Initializing matrix H...

                parameters:
```

```
               algorithm: HALS
      stopping criterion: Ratio of Projected Gradients
                  height: 12411
                   width: 7984
                       k: 16
                miniter: 5
                maxiter: 5000
                    tol: 0.005
              matrixfile: ../data/reuters.mtx
              maxthreads: 8

1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.710219
7:    progress metric:  0.580951
8:    progress metric:  0.471557
9:    progress metric:  0.491855
10:   progress metric:  0.531999
11:   progress metric:  0.353302
12:   progress metric:  0.201634
13:   progress metric:  0.1584
14:   progress metric:  0.142572
15:   progress metric:  0.12588
16:   progress metric:  0.113239
17:   progress metric:  0.0976934
18:   progress metric:  0.0821207
19:   progress metric:  0.0746089
20:   progress metric:  0.0720616
40:   progress metric:  0.0252854
60:   progress metric:  0.0142085
80:   progress metric:  0.0153269

Solution converged after 88 iterations.

Elapsed wall clock time: 1.560 sec.

Writing output files...

************************************************************
*                                                          *
*       Running NMF-RANK2 with W and H initializers        *
*                                                          *
************************************************************
Initializing matrix W...
Initializing matrix H...

                parameters:

               algorithm: Rank 2
      stopping criterion: Ratio of Projected Gradients
                  height: 12411
                   width: 7984
                       k: 2
                miniter: 5
                maxiter: 5000
                    tol: 0.005
              matrixfile: ../data/reuters.mtx
              maxthreads: 8

1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.0374741
7:    progress metric:  0.0252389
8:    progress metric:  0.0169805
9:    progress metric:  0.0113837
10:   progress metric:  0.00761077
11:   progress metric:  0.0050782
12:   progress metric:  0.00338569

Solution converged after 12 iterations.
```

```
Elapsed wall clock time: 0.028 sec.

Writing output files...

************************************************************
*                                                          *
*       Repeating the previous run with tol = 1.0e-5       *
*                                                          *
************************************************************
Initializing matrix W...
Initializing matrix H...

                parameters:

               algorithm: Rank 2
      stopping criterion: Ratio of Projected Gradients
                  height: 12411
                   width: 7984
                       k: 2
                miniter: 5
                maxiter: 5000
                    tol: 1e-05
              matrixfile: ../data/reuters.mtx
              maxthreads: 8

1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.0374741
7:    progress metric:  0.0252389
8:    progress metric:  0.0169805
9:    progress metric:  0.0113837
10:   progress metric:  0.00761077
11:   progress metric:  0.0050782
12:   progress metric:  0.00338569
13:   progress metric:  0.00225761
14:   progress metric:  0.00150429
15:   progress metric:  0.00100167
16:   progress metric:  0.000666691
17:   progress metric:  0.000443654
18:   progress metric:  0.000295213
19:   progress metric:  0.000196411
20:   progress metric:  0.000130604

Solution converged after 27 iterations.

Elapsed wall clock time: 0.061 sec.

Writing output files...
Minimum value in W matrix: 0.
Maximum value in W matrix: 0.397027.

************************************************************
*                                                          *
*      Running HierNMF2 with 5 clusters, JSON format       *
*                                                          *
************************************************************
loading dictionary...
creating random W initializers...
creating random H initializers...

              parameters:

                  height: 12411
                   width: 7984
              matrixfile: ../data/reuters.mtx
                dictfile: ../data/reuters_dictionary.txt
                     tol: 0.0001
                miniter: 5
                maxiter: 5000
                maxterms: 5
              maxthreads: 8
[1] [2] [3] [4]
```

```
Elapsed wall clock time: 391 ms.
9/9 factorizations converged.

Writing output files...

************************************************************
*                                                          *
* Running HierNMF2 with 10 clusters, 12 terms, XML format  *
*                                                          *
************************************************************
creating random W initializers...
creating random H initializers...

              parameters:

                  height: 12411
                   width: 7984
              matrixfile: ../data/reuters.mtx
                dictfile: ../data/reuters_dictionary.txt
                     tol: 0.0001
                miniter: 5
                maxiter: 5000
                maxterms: 12
              maxthreads: 8
[1] [2] [3] [4] [5] [6] dropping 20 items ...
[7] [8] [9]

Elapsed wall clock time: 837 ms.
21/21 factorizations converged.

Writing output files...

************************************************************
*                                                          *
*  Running HierNmf2 with 18 clusters, 8 terms, with flat   *
*                                                          *
************************************************************
creating random W initializers...
creating random H initializers...

              parameters:

                  height: 12411
                   width: 7984
              matrixfile: ../data/reuters.mtx
                dictfile: ../data/reuters_dictionary.txt
                     tol: 0.0001
                miniter: 5
                maxiter: 5000
                maxterms: 8
              maxthreads: 8
[1] [2] [3] [4] [5] [6] dropping 20 items ...
[7] [8] [9] dropping 25 items ...
[10] [11] [12] [13] [14] [15] [16] [17]

Running NNLS solver...
1:    progress metric:  1
2:    progress metric:  0.264152
3:    progress metric:  0.0760648
4:    progress metric:  0.0226758
5:    progress metric:  0.00743562
6:    progress metric:  0.00280826
7:    progress metric:  0.00103682
8:    progress metric:  0.000361738
9:    progress metric:  0.000133087
10:   progress metric:  5.84849e-05

Elapsed wall clock time: 1.362 s.
40/40 factorizations converged.

Writing output files...
```

IV.4 Matrix File Formats

The SmallK software supports comma-separated value (CSV) files for dense matrices and Matrix Market files for sparse matrices.

For example, the 5x3 dense matrix

```
42    47    52
43    48    53
44    49    54
45    50    55
46    51    56
```

would be stored in a CSV file as follows:

```
42,47,52
43,48,53
44,49,54
45,50,55
46,51,56
```

The matrix is loaded exactly as it appears in the file. **Internally, SmallK stores dense matrices in column-major order**. Sparse matrices are stored in **compressed column format**.

IV.5 SmallK API

The SmallK API is an extremely **simplistic** API for basic NMF and clustering. Users who require more control over the factorization or clustering algorithms can instead run one of the command-line applications in the SmallK distribution.

The SmallK API is exposed by the file `smallk.hpp`, which can be found in this location: `SMALLK_INSTALL_DIR/include/smallk.hpp`. All API functions are contained within the `smallk` namespace.

An example of how to use the API can be found in the file `examples/smallk_example.cpp`.

The smallk library maintains a set of state variables that are used to control the Nmf and clustering routines. Once set, the state variables maintain their values until changed by an API function. For instance, one state variable represents the matrix to be factored (or used for clustering). The API provides a function to load this matrix; once loaded, it can be repeatedly factored without the need for reloading. The state variables and their default values are documented below.

All computations with the smallk library are performed in double precision.

Enumerations

The SmallK API provides two enumerated types, one for the supported NMF algorithms and one for the clustering file output format. These are:

```
enum Algorithm
{
    MU,       // Multiplicative Updating, Lee & Seung
    BPP,      // Block Principal Pivoting, Kim and Park
    HALS,     // Hierarchical Alternating Least Squares, Cichocki & Pan
    RANK2     // Rank2, Kuang and Park
};
```

The default NMF algorithm is BPP. The Rank2 algorithm is optimized for two-column or two-row matrices and is the underlying factorization routine for the clustering code.

```
enum OutputFormat
{
    XML,  // Extensible Markup Language
    JSON  // JavaScript Object Notation
};
```

API functions

Initialization and Cleanup

```
void Initialize(int& argc,     // in
                char**& argv)  // in
```

Call this function first, before all others in the API; initializes Elemental and the smallk library.

```
bool IsInitialized()
```

Returns true if the library has been initialized via a call to `Initialize()`, false otherwise.

```
void Finalize()
```

Call this function last, after all others in the API; performs cleanup for Elemental and the smallk library.

Versioning

```
unsigned int GetMajorVersion()
```

Returns the major release version number of the library as an unsigned integer.

```
unsigned int GetMinorVersion()
```

Returns the minor release version number of the library as an unsigned integer.

```
unsigned int GetPatchLevel()
```

Returns the patch version number of the library as an unsigned integer.

```
std::string GetVersionString()
```

Returns the version of the library as a string, formatted as `major.minor.patch`.

Common Functions

```
unsigned int GetOutputPrecision()
```

Returns the floating point precision with which numerical output will be written (i.e., the computed W and H matrix factors from the `Nmf` routine). The default precision is six digits.

```
void SetOutputPrecision(const unsigned int num_digits)
```

Sets the floating point precision with which numerical output will be written. Input values should be within the range [1, precision(`double`)]. Any inputs outside of this range will be adjusted.

```
unsigned int GetMaxIter()
```

Returns the maximum number of iterations allowed for NMF computations. The default value is 5000.

```
void SetMaxIter(const unsigned int max_iterations = 5000)
```

Sets the maximum number of iterations allowed for NMF computations. The default of 5000 should be more than sufficient for most computations.

```
unsigned int GetMinIter()
```

Returns the minimum number of NMF iterations. The default value is 5.

```
void SetMinIter(const unsigned int min_iterations = 5)
```

Sets the minimum number of NMF iterations to perform before checking for convergence. The convergence and progress estimation routines are non-trivial calculations, so increasing this value may result in faster performance.

```
unsigned int GetMaxThreads()
```

Returns the maximum number of threads used for NMF or clustering computations. The default value is hardware-dependent, but is generally the maximum number allowed by the hardware.

```
void SetMaxThreads(const unsigned int max_threads);
```

Sets an upper limit to the number of threads used for NMF and clustering computations. Inputs that exceed the capabilities of the hardware will be adjusted. This function is provided for scaling and performance studies.

```
void Reset()
```

Resets all state variables to their default values.

```
void SeedRNG(const int seed)
```

Seeds the random number generator (RNG) within the smallk library. Normally this RNG is seeded from the system time whenever the library is initialized. The RNG is the '19937' Mersenne Twister implementation provided by the C++ standard library.

```
void LoadMatrix(const std::string& filepath)
```

Loads a matrix contained in the given file. The file must either be a comma-separated value (.CSV) file for a dense matrix, or a MatrixMarket-format file (.MTX) for a sparse matrix. If the matrix cannot be loaded the library throws a `std::runtime_error` exception.

```
bool IsMatrixLoaded()
```

Returns true if a matrix is currently loaded, false if not.

```
std::string GetOuputDir()
```

Returns a string indicating the directory into which output files will be written. The default is the current directory.

```
void SetOutputDir(const std::string& outdir)
```

Sets the directory into which output files should be written. The 'outdir' argument can either be an absolute or relative path. The default is the current directory.

NMF Functions

```
void Nmf(const unsigned int k,
         const Algorithm algorithm     = Algorithm::BPP,
         const std::string& initfile_w = std::string(""),
         const std::string& initfile_h = std::string(""))
```

This function nonnegatively factors the loaded input matrix A as follows: A ~ WH. If a matrix is not currently loaded a `std::logic_error` exception will be thrown. The default algorithm is NMF-BPP; provide one of the enumerated algorithm values to use a different algorithm.

Matrix A has dimension mxn; matrix W has dimension mxk; matrix H has dimension kxn. The value of k is provided as an argument.

Optional initializer matrices can be provided for the W and H factors via the 'initfile_w' and 'initfile_h' arguments. These files must contain fully dense matrices in .CSV format. The W matrix initializer must have dimension mxk, and the H matrix initializer must have dimension kxn. If the initializer matrices do not match these dimensions exactly a `std::logic_error` exception is thrown. If initializers are not provided, matrices W and H will be randomly initialized.

The computed factors W and H will be written to the output directory in the files 'w.csv' and 'h.csv'.

Exceptions will be thrown (either from Elemental or smallk) in case of error.

```
const double* LockedBufferW(unsigned int& ldim, unsigned int& height, unsigned int& width)
```

This function returns a READONLY pointer to the buffer containing the W factor computed by the `Nmf` routine, along with buffer and matrix dimensions. The 'ldim', 'height', and 'width' arguments are all *out* parameters. The buffer has a height of 'ldim' and a width of 'width'. The matrix W has the same width but a height of 'height', which may differ from ldim. The W matrix is stored in the buffer in column-major order. See the `examples/smallk_example.cpp` file for an illustration of how to use this function.

```
const double* LockedBufferH(unsigned int& ldim, unsigned int& height, unsigned int& width)
```

Same as LockedBufferW, but for the H matrix.

```
double GetNmfTolerance()
```

Returns the tolerance value used to determine NMF convergence. The default value is 0.005.

```
void SetNmfTolerance(const double tol=0.005)
```

Sets the tolerance value used to determine NMF convergence. The NMF algorithms are iterative, and at each iteration a progress metric is computed and compared with the tolerance value. When the metric falls below the tolerance value the iterations stop and convergence is declared. The tolerance value should satisfy `0.0 < tolerance < 1.0`. Any inputs outside this range will cause a `std::logic_error` exception to be thrown.

Clustering Functions

```
void LoadDictionary(const std::string& filepath)
```

Loads the dictionary used for clustering. The dictionary is an ASCII file of text strings as described in the preprocessor input files section below. If the dictionary file cannot be loaded a `std::runtime_error` exception is thrown.

```
unsigned int GetMaxTerms()
```

Returns the number of highest-probability dictionary terms to store per cluster. The default value is 5.

```
void SetMaxTerms(const unsigned int max_terms = 5)
```

Sets the number of highest-probability dictionary terms to store per cluster.

```
OutputFormat GetOutputFormat()
```

Returns a member of the `OutputFormat` enumerated type; this is the file format for the clustering results. The default output format is JSON.

```
void SetOutputFormat(const OutputFormat = OutputFormat::JSON)
```

Sets the output format for the clustering result file. The argument must be one of the values in the `OutputFormat` enumerated type.

```
double GetHierNmf2Tolerance()
```

Returns the tolerance value used by the NMF-RANK2 algorithm for hierarchical clustering. The default value is 1.0e-4.

```
void SetHierNmf2Tolerance(const double tol=1.0e-4)
```

Sets the tolerance value used by the NMF-RANK2 algorithm for hierarchical clustering. The tolerance value should satisfy `0.0 < tolerance < 1.0`. Any inputs outside this range will cause a `std::logic_error` exception to be thrown.

```
void HierNmf2(const unsigned int num_clusters)
```

This function performs hierarchical clustering on the loaded matrix, generating the number of clusters specified by the 'num_clusters' argument. For an overview of the hierarchical clustering process, see the description below for the hierclust command line application.

This function generates two output files in the output directory: 'assignments_N.csv' and 'tree_N.{json, xml}'. Here N is the number of clusters specified as an argument, and the tree file can be in either JSON XML format.

The content of the files is described below in the section on the hierclust command line application.

```
void HierNmf2WithFlat(const unsigned int num_clusters)
```

This function performs hierarchical clustering on the loaded matrix, exactly as described for `HierNmf2`. In addition, it also computes a flat clustering result. Thus four output files are generated. The flat clustering result files are 'assignments_flat_N.csv' and 'clusters_N.{json, xml}'. The cluster file contents are documented below in the section on the flatclust command line application.

V. SmallK Command Line Tools

The SmallK library provides a number of algorithm implementations for performing various data analytics tasks such as topic modeling, clustering, and dimension reduction. This section will provide more in-depth description of the tools available with examples that can be expanded/modified for other application domains.

Before diving into the various tools, it will be helpful to set up the command line environment to easily run the various executables that comprise the SmallK library. First the command line needs to know where to find the executable files to run the tools. Since while installing SmallK `'make_install'` was run, the executables are located in /usr/local/smallk/bin. Thus, this should be added to the `'$PATH'` system variable or added to the environment. The following command line performs the task of modifying the path avoiding the need to cd into directories were the tools are located:

```
export PATH=/usr/local/smallk/bin:$PATH
```

This allows the tools to be executed from any directory.

V.1. Preprocessor

Overview

The preprocessor prunes rows and columns from term-frequency matrices, attempting to generate a result matrix that is more suitable for clustering. It also computes tf-idf (term frequency-inverse document frequency) weights for the remaining entries. Therefore the input matrix consists of nonnegative integers, and the output matrix consists of floating point numbers between 0.0 and 1.0. The Matrix Market file format (.mtx file) is used for the input and output matrices.

Rows (terms) are pruned if a given term appears in fewer than 'DOCS_PER_TERM' documents. The value of DOCS_PER_TERM is a command-line parameter with a default value of 3. For a term-frequency input matrix, in which the matrix elements represent occurrence counts for the terms, this parameter actually specifies the minimum row sum for each term. Any rows whose row sums are less than this value will be pruned.

Columns (docs) are pruned if a given document contains fewer than 'TERMS_PER_DOC' terms. The value of TERMS_PER_DOC is a command-line parameter with a default value of 5.

Whenever columns (documents) are pruned the preprocessor checks the remaining columns for uniqueness. Any duplicate columns are identified and a representative column is chosen as the survivor. The code always selects the column with the largest column index in such groups as the survivor. The preprocessor continues to prune rows and columns until it finds no further candidates for pruning. It then computes new tf-idf scores for the resulting entries and writes out the result matrix in Matrix Market format.

If the preprocessor should prune all rows or columns, it writes an error message to the screen and terminates without generating any output.

Input Files

The preprocessor requires three input files: a matrix file, a dictionary file, and a document file. The matrix file contains a sparse matrix in Matrix Market format (.mtx). This is a term-frequency matrix, and all entries should be positive integers. The preprocessor can also read in matrices containing floating-point inputs, but only if 'boolean mode' is enabled; this will be described below. The preprocessor does not support dense matrices, since the typical matrices encountered in topic modeling problems are extremely sparse, with occupancies generally less than 1%.

The second file required by the preprocessor is a 'dictionary file'. This is a simple ASCII text file containing one entry per line. Entries represent keywords, bigrams, or other general text strings the user is interested in. Each line of the file is treated as a 'keyword'; so multi-word keywords are supported as well. The smallk/data folder contains a sample dictionary file called 'dictionary.txt'. The first few entries are:

```
triumph
dey
canada
finger
circuit
…
```

The third file required by the preprocessor is a 'documents file'. This is another simple ASCII text file containing one entry per line. Entries represent document names or other unique identifiers. The smallk/data folder also contains a sample documents file called 'documents.txt'. The first few entries of this file are:

```
52828-11101.txt
51820-10202.txt
104595-959.txt
60259-3040.txt
…
```

These are the unique document identifiers for the user who generated the file. Your identifiers will likely have a different format.

Finally, the preprocessor **requires** these files to have the following names: matrix.mtx, dictionary.txt, and documents.txt. The input folder containing these files can be specified on the command line (described below). The output of the preprocessor is a new set of files called 'reduced_matrix.mtx', 'reduced_dictionary.txt', and 'reduced_documents.txt'.

Command Line Options

The preprocessor binary is called 'preprocess_tf', to emphasize the fact that it operates on term-frequency matrices. If the binary is run with no arguments, it prints out the following information:

```
preprocess_tf
   --indir <path>
  [--outdir (defaults to current directory)]
  [--docs_per_term 3]
  [--terms_per_doc 5]
  [--maxiter 1000]
  [--precision 4]
  [--boolean_mode 0]
```

Only the first parameter, --indir, is required. All remaining parameters are optional and have the default values indicated.

The meanings of the various options are as follows:

1. `--indir`: path to the folder containing the files 'matrix.mtx', 'dictionary.txt', and 'documents.txt'
2. `--outdir`: path to the folder to into which results should be written
3. `--docs_per_term`: any rows whose entries sum to less than this value will be pruned
4. `--terms_per_doc`: any columns whose entries sum to less than this value will be pruned
5. `--maxiter`: perform no more than this many iterations
6. `--precision`: the number of digits of precision with which to write the output matrix
7. `--boolean_mode`: all nonzero matrix elements will be treated as if they had the value 1.0. In other words, the preprocessor will ignore the actual frequency counts and treat all nonzero entries as if they were 1.0.

Sample Runs

Here is a sample run of the preprocessor using the data provided in the smallk distribution. This run was performed from the top-level smallk folder after building the code:

```
preprocess_tf --indir data

      Command line options:

                  indir: data/
```

```
                 outdir: current directory
                 docs_per_term: 3
                 terms_per_doc: 5
                 max_iter: 1000
                 precision: 4
                 boolean_mode: 0

Loading input matrix data/matrix.mtx
      Input file load time: 1.176s.

Starting iterations...
      [1] height: 39771, width: 11237, nonzeros: 877453
Iterations finished.
      New height: 39727
      New width: 11237
      New nonzero count: 877374
Processing time: 0.074s.

Writing output matrix reduced_matrix.mtx
Output file write time: 2.424s.
Writing dictionary file reduced_dictionary.txt
Writing documents file reduced_documents.txt
Dictionary + documents write time: 0.08s.
```

V.2. Matrixgen

Overview

The matrix generator application is a simple tool for generating simple matrices. The NMF and clustering tools for various testing scenarios can load these matrices. Use of the matrix generator is entirely optional.

Command Line Options

Running the matrixgen binary with no options generates the following output:

```
matrixgen

Usage: matrixgen
         --height <number of rows>
         --width  <number of cols>
         --filename <path>
        [--type  UNIFORM]  UNIFORM:     matrix with uniformly-distributed random entries
                           DENSE_DIAG:  dense diagonal matrix with uniform random entries
                           SPARSE_DIAG: sparse diagonal matrix with uniform random entries
                           IDENTITY:    identity matrix
                           ONES:        matrix of all ones
                           ZEROS:       matrix of all zeros
                           SPARSE:      sparse matrix with uniform random entries
                                        specify 'nz_per_col' to control occupancy

        [--rng_center  0.5]   center of random numbers
        [--rng_radius  0.5]   radius of random numbers
        [--precision   6]     digits of precision
        [--nz_per_col  1]     (SPARSE only) nonzeros per column
```

The `--height`, `--width`, and `--filename` options are required. All others are optional and have the default values indicated.

The meanings of the various options are as follows:

1. `--height`: number of rows in the generated matrix
2. `--width`: number of columns in the generated matrix
3. `--filename`: name of the output file
4. `--type`: the type of matrix to be generated; the default is a uniformly-distributed random matrix
5. `--rng_center`: random number distribution will be centered on this value
6. `--rng_radius`: random numbers will span this distance to either side of the center value
7. `--precision`: the number of digits of precision with which to write the output matrix
8. `--nz_per_col`: number of nonzero entries per sparse matrix column; valid only for SPARSE type

Sample Runs

Suppose we want to generate a matrix of uniformly distributed random numbers. The matrix should have a height of 100 and a width of 16, and should be written to a file called 'w_init.csv'. Use the matrix generator as follows:

```
matrixgen --height 100 --width 16 --filename w_init.csv
```

V.3. Nonnegative Matrix Factorization (NMF)

Overview

The NMF command line application performs nonnegative matrix factorization on dense or sparse matrices. If the input matrix is denoted by A, nonnegative matrix factors W and H are computed such that $A \sim WH$. Matrix A can be either dense or sparse; matrices W and H are always dense. Matrix A has m rows and n columns; matrix W has m rows and k columns; matrix H has k rows and n columns. Parameter k is a positive integer and is typically much less than either m or n.

Command Line Options

Running the nmf application with no command line parameters will cause the application to display all params that it supports. These are:

```
Usage: nmf
        --matrixfile <filename>  Filename of the matrix to be factored.
                                 Either CSV format for dense or MatrixMarket format for sparse.
        --k <integer value>      The common dimension for factors W and H.
        [--algorithm  BPP]       NMF algorithms:
                                     MU:    multiplicative updating
                                     HALS:  hierarchical alternating least squares
                                     RANK2: rank2 with optimal active set selection
                                     BPP:   block principal pivoting
        [--stopping  PG_RATIO]   Stopping criterion:
                                     PG_RATIO: Ratio of projected gradients
                                     DELTA:    Change in relative F-norm of W
        [--tol  0.005]           Tolerance for the selected stopping criterion.
        [--tolcount  1]          Tolerance count; declare convergence after this many
                                 iterations with metric < tolerance; default is to
                                 declare convergence on the first such iteration.
        [--infile_W  (empty)]    Dense mxk matrix to initialize W; CSV file.
                                 If unspecified, W will be randomly initialized.
        [--infile_H  (empty)]    Dense kxn matrix to initialize H; CSV file.
                                 If unspecified, H will be randomly initialized.
        [--outfile_W  w.csv]     Filename for the W matrix result.
        [--outfile_H  h.csv]     Filename for the H matrix result.
        [--miniter  5]           Minimum number of iterations to perform.
        [--maxiter  5000]        Maximum number of iterations to perform.
        [--outprecision  6]      Write results with this many digits of precision.
        [--maxthreads  8]        Upper limit to thread count.
        [--normalize  1]         Whether to normalize W and scale H.
                                     1 == yes, 0 == no
        [--verbose  1]           Whether to print updates to the screen.
                                     1 == print updates, 0 == silent
```

The `--matrixfile` and `--k` options are required; all others are optional and have the default values indicated. The meanings of the various options are as follows:

1. `--matrixfile`: Filename of the matrix to be factored. CSV files are supported for dense matrices and MTX files for sparse matrices.
2. `--k`: the width of the W matrix (identical to the height of the H matrix)
3. `--algorithm`: identifier for the factorization algorithm
4. `--stopping`: the method used to terminate the iterations; use PG_RATIO unless you have a specific reason not to
5. `--tol`: tolerance value used to terminate iterations; when the progress metric falls below this value iterations will stop; typical values are in the 1.0e-3 or 1.0e-4 range
6. `--tolcount`: a positive integer representing the number of successive iterations for which the progress metric must have a value <= tolerance; default is 1, which means the iterations will terminate on the first iteration with progress_metric <= tolerance
7. `--infile_W`: CSV file containing the mxk initial values for matrix W; if omitted, W is randomly initialized
8. `--infile_H`: CSV file containing the kxn initial values for matrix H; if omitted, H is randomly initialized
9. `--outfile_W`: filename for the computed W factor; default is w.csv

10. `--outfile_H`: filename for the computed H factor; default is h.csv
11. `--miniter`: the minimum number of iterations to perform before checking progress; for smaller tolerance values, you may want to increase this number to avoid needless progress checks
12. `--maxiter`: the maximum number of iterations to perform
13. `--outprecision`: matrices W and H will be written to disk using this many digits of precision
14. `--maxthreads`: the maximum number of threads to use; the default is to use as many threads as the hardware can support (your number may differ from that shown)
15. `--normalize`: whether to normalize the columns of the W matrix and correspondingly scale the rows of H after convergence
16. `--verbose`: whether to display updates to the screen as the iterations progress

Sample Runs

The smallk distribution contains a 'data' directory with a matrix file 'reuters.mtx'. This is a tf-idf weighted matrix derived from the popular Reuters data set used in machine learning experiments.

Suppose we want to factor the Reuters matrix using a k value of 8. We would do that as follows, assuming that we are in the top-level smallk folder after building the code:

```
nmf/bin/nmf --matrixfile data/reuters.mtx  --k 8
```

If we want to instead use the HALS algorithm with k=16, a tolerance of 1.0e-4, and also perform 10 iterations prior to checking progress, we would use this command line:

```
nmf/bin/nmf --matrixfile data/reuters.mtx --k 16 --algorithm HALS --tol 1.0e-4 --miniter 10
```

To repeat the previous experiment but with new names for the output files, we would do this:

```
nmf/bin/nmf --matrixfile data/reuters.mtx --k 16 --algorithm HALS --tol 1.0e-4
--miniter 10 --outfile_W w_hals.csv -outfile_H h_hals.csv
```

V.4. Hierclust

Overview

First, we briefly describe the algorithm and the references section provides pointers to papers with detailed descriptions of the algorithms. NMF-RANK2 for hierarchical clustering generates a binary tree of items. We refer to a node in the binary tree and the items associated with the node interchangeably. This method begins by placing all data items in the root node. The number of leaf nodes to generate is specified (user input). The algorithm proceeds with the following steps, repeated until the maximum number of leaf nodes, max_leaf_nodes, is reached:

1. Pick the leaf node with the highest score (at the very beginning where only a root node is present, just pick the root node)
2. Apply NMF-RANK2 to the node selected in step 1, and generate two new leaf nodes
3. Compute a score for each of the two leaf nodes generated in step 2
4. Repeat until the desired number of leaf nodes has been generated

Step 2 implements the details of the node splitting into child nodes. Outlier detection plays a crucial role in hierarchical clustering to generate a tree with well-balanced and meaningful clusters. To implement this, we have two additional parameters in step 2: *trial_allowance* and *unbalanced*.

The parameter *trial_allowance* is the number of times that the program will try to split a node into two meaningful clusters. In each trial, the program will check if one of the two generated leaf nodes is an outlier set. If the outlier set is detected, the program will delete the items in the outlier set from the node being split and continue to the next trial. If all the trials are finished and the program still cannot find two meaningful clusters for this node, all the deleted items are "recycled" and placed into this node again, and this node will be labeled as a "permanent leaf node" that cannot be picked in step 1 in later iterations.

The parameter *unbalanced* is a threshold parameter to determine whether two generated leaf nodes are unbalanced. Suppose two potential leaf nodes L and R are generated from the selected node and L has fewer items than R. Let us denote the number of items in a node N as |N|. L and R are called *unbalanced* if |L| < unbalanced * (|L|+|R|). Note that if L and R are unbalanced, the potential node L with fewer items is not necessarily regarded as an outlier set. Please see the referenced paper for more details [3].

Internally, NMF-RANK2 is applied to each leaf node to compute the score in step 3. The computed result matrices W and H in step 3 are cached so that we can avoid duplicate work in step 2 in later iterations.

The score for each leaf node is based on a modified version of the NDCG (*Normalized Discounted Cumulative Gain*) measure, a common measure in the information retrieval community. A leaf node is associated with a "topic vector", and we can define "top terms" based on the topic vector. A leaf node will receive a high score if its top terms are a good combination of the top terms of its two potential children; otherwise it receives a low score.

The hierclust application generates two output files. One file contains the assignments of documents to clusters. This file contains one integer for each document (column) of the original matrix. The integers are the cluster labels for that cluster that the document was assigned to. If the document could not be assigned to a cluster, a -1 will be entered into the file, indicating that the document is an outlier.

The other output file contains information for each node in the factorization binary tree. The items in this file are:

1. `id`: a unique id for this node
2. `level`: the level in the tree at which this node appears; the root is at level 0, the children of the root are at level 1, etc.
3. `label`: the cluster label for this node (meaningful only for leaf nodes)
4. `parent_id`: the unique id of the parent of this node (the root node has parent_id == 0)
5. `parent_label`: the cluster label of the parent of this node

6. `left_child`: a Boolean value indicating whether this node is the left or right child of its parent
7. `left_child_label`: the cluster label of the left child of this node (leaf nodes have -1 for this value)
8. `right_child_label`: the cluster label of the right child of this node (leaf nodes have -1 for this value)
9. `doc_count`: the number of documents that this node represents
10. `top_terms`: the highest probability dictionary terms for this node

The node id values and the left or right child indicators can be used to unambiguously reconstruct the factorization tree.

Command Line Options

Running the hierclust application with no command line parameters will cause the application to display all params that it supports. These are:

```
hierclust

Usage: hierclust
        --matrixfile <filename>     Filename of the matrix to be factored.
                                    Either CSV format for dense or MatrixMarket format for sparse.
        --dictfile <filename>       The name of the dictionary file.
        --clusters <integer>        The number of clusters to generate.
        [--infile_W  (empty)]       Dense m x (4*clusters) matrix to initialize W, CSV file.
                                    If unspecified, W will be randomly initialized.
        [--infile_H  (empty)]       Dense (4*clusters) x n matrix to initialize H, CSV file.
                                    If unspecified, H will be randomly initialized.
        [--tol  0.0001]             Tolerance value for each factorization.
        [--outdir  (empty)]         Output directory.  If unspecified, results will be
                                    written to the current directory.
        [--miniter  5]              Minimum number of iterations to perform.
        [--maxiter  5000]           Maximum number of  iterations to perform.
        [--maxterms  5]             Number of terms per node.
        [--maxthreads    8]         Upper limit to thread count.
        [--unbalanced  0.1]         Threshold for determining leaf node imbalance.
        [--trial_allowance  3]      Number of split attempts.
        [--flat  0]                 Whether to generate a flat clustering result.
                                        1 == yes, 0 == no
        [--verbose  1]              Whether to print updates to the screen.
                                        1 == yes, 0 == no
        [--format  XML]             Format of the output file containing the tree.
                                        XML: XML format
                                        JSON: JavaScript Object Notation
        [--treefile  tree_N.ext]    Name of the output file containing the tree.
                                    N is the number of clusters for this run.
                                    The string 'ext' depends on the desired format.
                                    This filename is relative to the outdir.
        [--assignfile assignments_N.csv]  Name of the file containing final assignments.
                                          N is the number of clusters for this run.
                                          This filename is relative to the outdir.
```

The `--matrixfile`, `--dictfile`, and `--clusters` options are required; all others are optional and have the default values indicated. The meanings of the various options are as follows:

1. `--matrixfile`: Filename of the matrix to be factored. CSV files are supported for dense matrices and MTX files for sparse matrices.
2. `--dictfile`: absolute or relative path to the dictionary file
3. `--clusters`: the number of leaf nodes (clusters) to generate
4. `--infile_W`: CSV file containing the mx(4*clusters) initial values for matrix W; if omitted, W is randomly initialized
5. `--infile_H`: CSV file containing the (4*clusters)xn initial values for matrix H; if omitted, H is randomly initialized
6. `--tol`: tolerance value for each internal NMF-RANK2 factorization; the stopping criterion is the ratio of projected gradient method
7. `--outdir`: path to the folder into which to write the output files; if omitted results will be written to the current directory
8. `--miniter`: minimum number of iterations to perform before checking progress on each NMF-RANK2 factorization
9. `--maxiter`: the maximum number of iterations to perform on each NMF-RANK2 factorization

10. `--maxterms`: the number of dictionary keywords to include in each node
11. `--maxthreads`: the maximum number of threads to use; the default is to use as many threads as the hardware can support (your number may differ from that shown)
12. `--unbalanced`: threshold value for declaring leaf node imbalance (see explanation above)
13. `--trial_allowance`: maximum number of split attempts for any node (see explanation above)
14. `--flat`: whether to generate a flat clustering result in addition to the hierarchical clustering result
15. `--verbose`: whether to display updates to the screen as the iterations progress
16. `--format`: file format to use for the clustering results
17. `--treefile`: name of the output file for the factorization tree; uses the format specified by the format parameter
18. `--assignfile`: name of the output file for the cluster assignments

Sample Runs

The smallk distribution contains a 'data' directory with a matrix file 'reuters.mtx' and an associated dictionary file 'reuters_dictionary.txt'. These files are derived from the popular Reuters data set used in machine learning experiments.

Suppose we want to perform hierarchical clustering on this data set and generate 10 leaf nodes. We would do that as follows, assuming that we are in the top-level smallk folder after building the code:

```
hierclust/bin/hierclust --matrixfile data/reuters.mtx  --dictfile data/reuters_dictionary.txt
--clusters 10
```

This will generate two result files in the current directory: tree_10.xml and assignments_10.csv.

If we want to instead generate 10 clusters, each with 8 terms, using JSON output format, we would use this command line:

```
hierclust/bin/hierclust --matrixfile data/reuters.mtx  --dictfile data/reuters_dictionary.txt
--clusters 10 --maxterms 8 --format JSON
```

Two files will be generated: tree_10.json and assignments_10.csv. The json file will have 8 keywords per node, whereas the tree_10.xml file will have only 5.

To generate a flat clustering result (in addition to the hierarchical clustering result), use this command line:

```
hierclust/bin/hierclust --matrixfile data/reuters.mtx  --dictfile data/reuters_dictionary.txt
--clusters 10 --maxterms 8 --format JSON --flat 1
```

Two additional files will be generated this time (along with tree_10.json and assignments_10.csv): 'clusters_10.json', which contains the flat clustering results, and 'assignments_flat_10.csv', which contains the flat clustering assignments.

V.5. Flatclust

Overview

The flatclust command line application factors the input matrix using either NMF-HALS or NMF-BPP and generates a flat clustering result. A flatclust run generating k clusters will generally run more slowly than a hierclust run, of the same number of clusters, with the `--flat` option enabled. The reason for this is that the hierclust application uses the NMF-RANK2 algorithm and always generates factor matrices with two rows or columns. The runtime of NMF scales super linearly with k, and thus runs fastest for the smallest k value.

The flatclust application generates two output files. The first file contains the assignments of documents to clusters and is interpreted identically to that of the hierclust application, with the exception that there are no outliers generated by flatclust.

The second file contains the node information. This file is much simpler than that of the hierclust application since there is no factorization tree. The items for each node in this file are:

1. `id`: the unique id of this node
2. `doc_count`: the number of documents assigned to this node
3. `top_terms`: the highest probability dictionary terms assigned to this node

Command Line Options

Running the flatclust application with no command line parameters will cause the application to display all params that it supports. These are:

```
flatclust

Usage: flatclust
        --matrixfile <filename>      Filename of the matrix to be factored.
                                     Either CSV format for dense or MatrixMarket format for sparse.
        --dictfile <filename>        The name of the dictionary file.
        --clusters <integer>         The number of clusters to generate.
        [--algorithm  BPP]           The NMF algorithm to use:
                                         HALS:  hierarchical alternating least squares
                                         RANK2: rank2 with optimal active set selection
                                                (for two clusters only)
                                         BPP:   block principal pivoting
        [--infile_W  (empty)]        Dense matrix to initialize W, CSV file.
                                     The matrix has m rows and 'clusters' columns.
                                     If unspecified, W will be randomly initialized.
        [--infile_H  (empty)]        Dense matrix to initialize H, CSV file.
                                     The matrix has 'clusters' rows and n columns.
                                     If unspecified, H will be randomly initialized.
        [--tol  0.0001]              Tolerance value for the progress metric.
        [--outdir  (empty)]          Output directory.  If unspecified, results will be
                                     written to the current directory.
        [--miniter  5]               Minimum number of iterations to perform.
        [--maxiter  5000]            Maximum number of  iterations to perform.
        [--maxterms  5]              Number of terms per node.
        [--maxthreads  8]            Upper limit to thread count.
        [--verbose  1]               Whether to print updates to the screen.
                                         1 == yes, 0 == no
        [--format  XML]              Format of the output file containing the tree.
                                         XML: XML format
                                         JSON: JavaScript Object Notation
        [--clustfile clusters_N.ext] Name of the output XML file containing the tree.
                                     N is the number of clusters for this run.
                                     The string 'ext' depends on the desired format.
                                     This filename is relative to the outdir.
        [--assignfile assignments_N.csv]  Name of the file containing final assignments.
                                          N is the number of clusters for this run.
                                          This filename is relative to the outdir.
```

The `--matrixfile`, `--dictfile`, and `--clusters` options are required; all others are optional and have the default values indicated. The meanings of the various options are as follows:

1. `--matrixfile`: Filename of the matrix to be factored. CSV files are supported for dense matrices and MTX files for sparse matrices.
2. `--dictfile`: absolute or relative path to the dictionary file
3. `--clusters`: the number of clusters to generate (equivalent to the NMF 'k' value)

4. `--algorithm`: the factorization algorithm to use
5. `--infile_W`: CSV file containing the m x 'clusters' initial values for matrix W; if omitted, W is randomly initialized
6. `--infile_H`: CSV file containing the 'clusters' x n initial values for matrix H; if omitted, H is randomly initialized
7. `--tol`: tolerance value for the factorization; the stopping criterion is the ratio of projected gradient method
8. `--outdir`: path to the folder into which to write the output files; if omitted results will be written to the current directory
9. `--miniter`: minimum number of iterations to perform before checking progress
10. `--maxiter`: the maximum number of iterations to perform
11. `--maxterms`: the number of dictionary keywords to include in each node
12. `--maxthreads`: the maximum number of threads to use; the default is to use as many threads as the hardware can support (your number may differ from that shown)
13. `--verbose`: whether to display updates to the screen as the iterations progress
14. `--format`: file format to use for the clustering results
15. `--clustfile`: name of the output file for the nodes; uses the format specified by the format parameter
16. `--assignfile`: name of the output file for the cluster assignments

Sample Run

The smallk distribution contains a 'data' directory with a matrix file 'reuters.mtx' and an associated dictionary file 'reuters_dictionary.txt'. These files are derived from the popular Reuters data set used in machine learning experiments.

Suppose we want to perform flat clustering on this data set and generate 10 clusters. We would do that as follows, assuming that we are in the top-level smallk folder after building the code:

```
flatclust/bin/flatclust --matrixfile data/reuters.mtx  --dictfile data/reuters_dictionary.txt
--clusters 10
```

This will generate two result files in the current directory: clusters_10.xml and assignments_10.csv.

If we want to instead generate 10 clusters, each with 8 terms, using JSON output format, we would use this command line:

```
flatclust/bin/flatclust --matrixfile data/reuters.mtx  --dictfile data/reuters_dictionary.txt
--clusters 10 --maxterms 8 --format JSON
```

Two files will be generated: clusters_10.json and assignments_10.csv. The json file will have 8 keywords per node, whereas the clusters_10.xml file will have only 5.

V.6. SmallK Test Results

After building the SmallK library, the 'make check' command will run a bash script that performs a series of tests on the code. Below is a sample output of those tests:

```
make check
sh tests/scripts/test_smallk.sh
**************************************************
*                                                *
*           Testing the smallk interface.        *
*                                                *
**************************************************
WARNING: Could not achieve THREAD_MULTIPLE support.
Smallk major version: 1
Smallk minor version: 0
Smallk patch level:   0
Smallk version string: 1.0.0
Loading matrix...

Running NMF-BPP...

Initializing matrix W...
Initializing matrix H...

               parameters:

              algorithm: Nonnegative Least Squares with Block Principal Pivoting
     stopping criterion: Ratio of Projected Gradients
                 height: 12411
                  width: 7984
                      k: 8
                miniter: 1
                maxiter: 5000
                    tol: 0.005
             matrixfile: data/reuters.mtx
             maxthreads: 8

1:    progress metric:  (min_iter)
2:    progress metric:  0.35826
3:    progress metric:  0.172127
4:    progress metric:  0.106297
5:    progress metric:  0.0696424
6:    progress metric:  0.0538889
7:    progress metric:  0.0559478
8:    progress metric:  0.0686117
9:    progress metric:  0.0788641
10:   progress metric:  0.0711522
11:   progress metric:  0.0493872
12:   progress metric:  0.0296543
13:   progress metric:  0.0189133
14:   progress metric:  0.0136437
15:   progress metric:  0.0103478
16:   progress metric:  0.00835688
17:   progress metric:  0.0071193
18:   progress metric:  0.00638383
19:   progress metric:  0.00596962
20:   progress metric:  0.00568349

Solution converged after 22 iterations.

Elapsed wall clock time: 0.485 sec.

Writing output files...

Running HierNmf2...

loading dictionary...
creating random W initializers...
creating random H initializers...

             parameters:

                 height: 12411
                  width: 7984
             matrixfile: data/reuters.mtx
               dictfile: data/reuters_dictionary.txt
                    tol: 0.0001
```

```
                miniter: 1
                maxiter: 5000
               maxterms: 5
             maxthreads: 8
[1] [2] [3] [4]

Elapsed wall clock time: 462 ms.
9/9 factorizations converged.

Writing output files...
W matrix test passed
H matrix test passed
**************************************************
*                                                *
*            Testing the preprocessor.           *
*                                                *
**************************************************

      Command line options:

                 indir: data/
                outdir: current directory
          docs_per_term: 3
          terms_per_doc: 5
               max_iter: 1000
              precision: 4
           boolean_mode: 0

Loading input matrix data/matrix.mtx
      Input file load time: 1.216s.

Starting iterations...
      [1] height: 39771, width: 11237, nonzeros: 877453
Iterations finished.
      New height: 39727
      New width: 11237
      New nonzero count: 877374
Processing time: 0.045s.

Writing output matrix reduced_matrix.mtx
Output file write time: 2.357s.
Writing dictionary file reduced_dictionary.txt
Writing documents file reduced_documents.txt
Dictionary + documents write time: 0.082s.
preprocessor matrix test passed
preprocessor dictionary test passed
preprocessor documents test passed
**************************************************
*                                                *
*            Testing the NMF routines.           *
*                                                *
**************************************************
WARNING: Could not achieve THREAD_MULTIPLE support.
Loading matrix...
Initializing matrix W...
Initializing matrix H...

      Command line options:

              algorithm: Rank 2
      stopping criterion: Ratio of Projected Gradients
                 height: 12411
                  width: 7984
                      k: 2
                miniter: 1
                maxiter: 5000
                    tol: 0.005
               tolcount: 1
                verbose: 1
              normalize: 1
           outprecision: 6
             matrixfile: data/reuters.mtx
               infile_W: data/nmf_init_w.csv
               infile_H: data/nmf_init_h.csv
              outfile_W: w.csv
              outfile_H: h.csv
             maxthreads: 8
```

```
1:    progress metric:  (min_iter)
2:    progress metric:  0.426546
3:    progress metric:  0.145075
4:    progress metric:  0.0852205
5:    progress metric:  0.0558555
6:    progress metric:  0.0374741
7:    progress metric:  0.0252389
8:    progress metric:  0.0169805
9:    progress metric:  0.0113837
10:   progress metric:  0.00761077
11:   progress metric:  0.0050782
12:   progress metric:  0.00338569

Solution converged after 12 iterations.

Elapsed wall clock time: 0.031 sec.

Writing output files...
NMF W matrix test passed
NMF H matrix test passed
**************************************************
*                                                *
*                Testing hierclust.              *
*                                                *
**************************************************
WARNING: Could not achieve THREAD_MULTIPLE support.
loading dictionary...
loading matrix...
loading W initializers...
loading H initializers...

     Command line options:

                 height: 12411
                  width: 7984
             matrixfile: data/reuters.mtx
               infile_W: data/hierclust_init_w.csv
               infile_H: data/hierclust_init_h.csv
               dictfile: data/reuters_dictionary.txt
             assignfile: assignments_5.csv
                 format: XML
               treefile: tree_5.xml
               clusters: 5
                    tol: 0.0001
                 outdir:
                miniter: 1
                maxiter: 5000
               maxterms: 5
             maxthreads: 8
             unbalanced: 0.1
        trial_allowance: 3
                   flat: 0
                verbose: 1

[1] [2] [3] [4]

Elapsed wall clock time: 583 ms.
9/9 factorizations converged.

Writing output files...
hierclust cluster file test passed
hierclust assignment file test passed
**************************************************
*                                                *
*              Testing flatclust.                *
*                                                *
**************************************************
WARNING: Could not achieve THREAD_MULTIPLE support.
loading dictionary...
loading matrix...
Initializing matrix W...
Initializing matrix H...

     Command line options:

                 height: 256
```

```
                width: 256
           matrixfile: data/rnd_256_256.csv
             infile_W: data/flatclust_init_w.csv
             infile_H: data/flatclust_init_h.csv
             dictfile: data/reuters_dictionary.txt
           assignfile: assignments_16.csv
               format: XML
            clustfile: clusters_16.xml
            algorithm: HALS
             clusters: 16
                  tol: 0.0001
               outdir:
              miniter: 1
              maxiter: 5000
             maxterms: 5
           maxthreads: 8
              verbose: 1

1:    progress metric:  (min_iter)
2:    progress metric:  0.635556
3:    progress metric:  0.490817
4:    progress metric:  0.479135
5:    progress metric:  0.474986
6:    progress metric:  0.44968
7:    progress metric:  0.422542
8:    progress metric:  0.407662
9:    progress metric:  0.395145
10:   progress metric:  0.379238
11:   progress metric:  0.36705
12:   progress metric:  0.35758
13:   progress metric:  0.350583
14:   progress metric:  0.343709
15:   progress metric:  0.336702
16:   progress metric:  0.328778
17:   progress metric:  0.318239
18:   progress metric:  0.302184
19:   progress metric:  0.286608
20:   progress metric:  0.272868
< many lines omitted>
2040: progress metric:  0.000157751
2060: progress metric:  0.000147217
2080: progress metric:  0.000137148
2100: progress metric:  0.000127922
2120: progress metric:  0.000119548
2140: progress metric:  0.000111997
2160: progress metric:  0.000104838

Solution converged after 2175 iterations.

Elapsed wall clock time: 0.997 sec.

XML file test passed
assignment file test passed
```

VI. SmallK Python Interface

Why Python? Although it's perfectly fine to run SmallK from the command line, Python provides a great deal more flexibility that augments the C++ code with other tasks that are much more easily accomplished with a very high level language. Python distributions can be easily extended with open source libraries from third part sources as well, two examples being numpy and scipy, well-known standards for scientific computing in the Python community. There are numerous packages available that extend these scientific libraries into the data analytics domain as well, such as scikit-learn.

We strongly recommend the Anaconda Python distribution provided by Continuum Analytics. Download and installation instructions for all platforms can be found here. Anaconda includes many if not most of the commonly used scientific and data analytics packages available and a very easy to use package manager and updating system. After installing Anaconda there will be available at the command line both a standard Python interpreter (type 'python') and an iPython interpreter (type 'ipython'). We recommend using the iPython interpreter. In addition to the command line interfaces to Python,

Anaconda includes the Spyder visual development environment featuring a very well thought out interface that makes developing Python code almost “too easy”. Spyder has many features found in the Matlab™ editor and a similar look and feel.

Anaconda also includes the Cython package, which is used by SmallK to integrate the Python and C++ code. Cython includes support for most of the C++ standard and supports the latest GNU C++ compilers. Most if not all the standard libraries are supported and the latest version (20.2) has support for the standard template library (STL) as well.

VI.1. Building the Python/Cython C++ Integration Libraries

Each of the components of the SmallK library can be built independently. Coming soon these will be incorporated into the SmallK build/installation system using the make tools and the SmallK Python tools will be made available as site package for the Python installation.

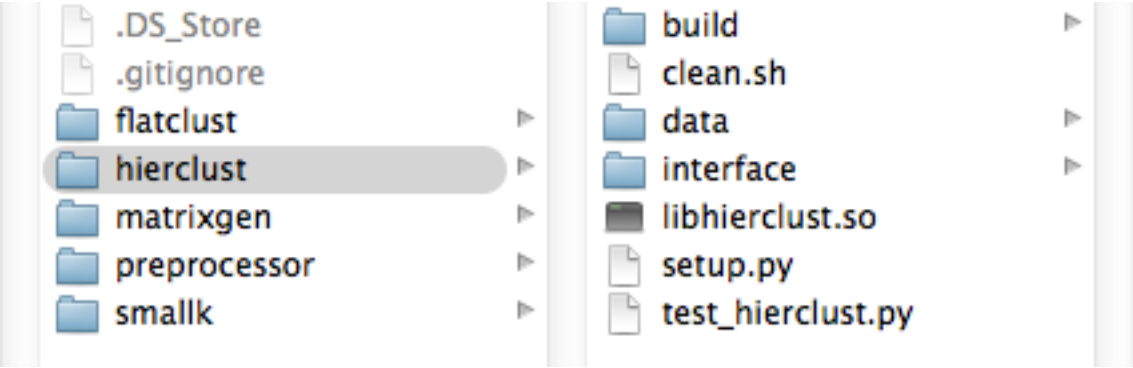
The components shared libraries that will be built correspond to the SmallK installed executables in /usr/local/smallk/bin: flatclust (libflatclust.so), hierclust (libhierclust.so), matrixgen (libmatrixgen.so), nmf (libsmallkpy.so), and preprocess_tf (libpreprocess.so). These will be located in /usr/local/smallk/lib after installation.

VI.1.1 Building each shared library

The SmallK Cython code can be found in <smallk_home> in the following directory:
<smallk_home> /pysmallk/cython

Where <smallk_home> is the directory in which you cloned the git repository or unpacked the tarball. All of what follows assumes that SmallK has been built and installed on your system. Under the cython directory will be found subdirectories corresponding to each of the component code packages of the SmallK C++ code.

The directory structure looks like the following figure on OSX using Finder:



In the above figure on the left pane is listed each of the components of SmallK. In the left pane ‘hierclust’ is selected revealing several files and directories in the right pane.

a. clean.sh – a shell script that cleans up the build
b. setup.py – the Python script used to build the installation
c. test_hierclust.py – a test script that imports libhierclust.so and calls C++ functions built into a static library located in /usr/local/smallk/lib (where libhierclust.so will be located for accessibility)

Each of the other directories on the left pane has the same structure and the same or similar files mentioned above.

The steps to build the Cython code are:

1. cd into the directory containing the setup.py file
2. build the shared library with the command line:

```
python setup.py build_ext --inplace
```

Do this for each of the subdirectories in the left pane.

Those two steps are all that are needed to build each of the Cython interfaces to the SmallK C++ functions used in the Python code. Change into each of the subdirectories in turn and perform the same two steps. Test each installation by running the test Python code in each directory. Here is an example command line and sample output for test_hierclust.py:

```
python test_hierclust.py --clusters 15 --matrixfile data/reuters.mtx --dictfile
data/reuters_dictionary.txt
```

```
Initialized:  True
Matrix loaded sucessfully True
Factor matrices W loaded initialized successfully True
Factor matrices W loaded initialized successfully True
1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.0449678
7:    progress metric:  0.0303641
8:    progress metric:  0.0204727
9:    progress metric:  0.0137662
10:   progress metric:  0.00922207
11:   progress metric:  0.00616192
12:   progress metric:  0.00410983
13:   progress metric:  0.00274049
14:   progress metric:  0.00182703
15:   progress metric:  0.00121719
16:   progress metric:  0.00081036
17:   progress metric:  0.000539294
18:   progress metric:  0.000358872
19:   progress metric:  0.000238792
20:   progress metric:  0.000158849

Solution converged after 22 iterations.

1:    progress metric:  (min_iter)
2:    progress metric:  (min_iter)
3:    progress metric:  (min_iter)
4:    progress metric:  (min_iter)
5:    progress metric:  (min_iter)
6:    progress metric:  0.0841783
7:    progress metric:  0.0967883
8:    progress metric:  0.0931827
9:    progress metric:  0.0682018
10:   progress metric:  0.0563845
11:   progress metric:  0.0539918
12:   progress metric:  0.0592587
13:   progress metric:  0.0695238
14:   progress metric:  0.0823972
15:   progress metric:  0.0900869
16:   progress metric:  0.0829633
17:   progress metric:  0.063792
18:   progress metric:  0.041053
19:   progress metric:  0.0253544
20:   progress metric:  0.0186906
40:   progress metric:  0.00137085
60:   progress metric:  0.000176433

Solution converged after 66 iterations.
```

```
...

NMF ran successfully True
Elapsed wall clock time:  1.16407084465 s
31 / 31  factorizations converged.
Writing output files... done.
```

After completing the build, copy (for now) each of the .so files into /usr/local/smallk/lib. Make this directory accessible from any where on your system by adding this directory to your path with this command line:

```
export PATH=/usr/local/smallk/lib:$PATH
```

VI.2. Python/Cython/C++ End Notes

Since this is ongoing work, the procedure for integrating the Python and C++ code will become more automated as we incorporate the Python/Cython build and installation into our C++ build/installation process.

Please check our website at smallk.github.io for the latest updates. Also, see the tutorial and quickstart document for examples of using the SmallK NMF library.

VII. References

References

1. J. Kim, Y. He, H. Park, *Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework*, Journal of Global Optimization, 2013.
2. J. Kim, and H. Park, *Sparse nonnegative matrix factorization for clustering*, Atlanta, GA, Report GT-CSE-08-01, Georgia Inst. of Technology, 2008.
3. D. Kuang and H. Park, *Fast rank-2 nonnegative matrix factorization for hierarchical document clustering*, KDD: Proc. of the 19th ACM Int. Conf. on Knowledge Discovery and Data Mining, 2013.

Contact Information

Richard Boyd
Senior Research Scientist
Cyber Technology and Information Security Lab
GA Tech Research Institute
250 14th St NW
Atlanta, GA 30318
richard.boyd@gtri.gatech.edu

Barry Drake
Senior Research Scientist
Cyber Technology and Information Security Lab
GA Tech Research Institute
250 14th St NW
Atlanta, GA 30318
barry.drake@gtri.gatech.edu