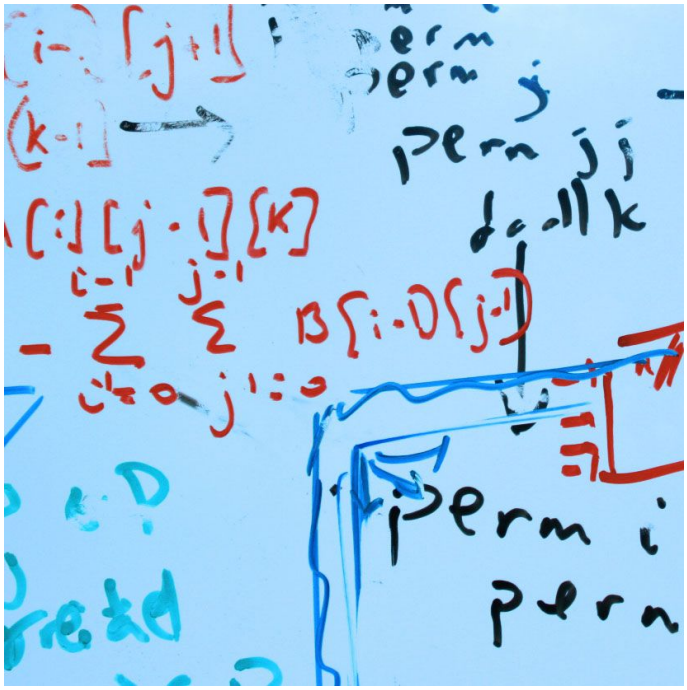
ENSIGN 4.2 User Guide

Project Name: **Exascale Non-Stationary Graph Notation (ENSIGN)**

Document Name: **ENSIGN User Guide**
Document Version: **4.2**

Reservoir Labs

Prepared by: **Reservoir Labs, Inc.**
632 Broadway, Suite 803
New York, NY 10012



Date: **February 11, 2020**

TERMS AND CONDITIONS

Table of Contents

1 Scope

1.1 Purpose

This document provides instructions for users to install and use the ENSIGN software package (version 4.2).

1.2 System Overview

ENSIGN is a software package for computing optimized sparse tensor decompositions.

The aim of ENSIGN is to move research on algorithms and methods for forming, decomposing, and visualizing tensors into tools that are suitable for experimentation by non-tensor experts (data scientists). ENSIGN is in continuous development and each release of ENSIGN embodies technology and features at varying stages of maturity. See Section 3.4 for the status of the decomposition methods.

The ENSIGN software package consists of two separate primary components:

- **Command-line decomposition methods**: tensor decomposition algorithms
- **Python command-line tools & API**: visualization, decomposition pre- and post-processing, and access to static decompositions through Python

ENSIGN is **TRL 6**.

1.3 Document Overview

This document provides requirements and procedures for installing ENSIGN and instructions and guidelines for using the tensor construction, decomposition, and visualization tools comprising the software package.

2 Installation Overview

2.1 Description

This section provides an overview of the software installation process for ENSIGN.

The following platforms are supported by this release:

- RHEL 7.6
- Ubuntu 16.04
- CentOS 7.2

2.2 Installation Prerequisites

The user is responsible for installing prerequisite software packages that are not included with the OS or the ENSIGN software package. The following table contains all package requirements for ENSIGN, the tested version, and the dependent components of ENSIGN.

PACKAGE	TESTED VERSION	NEEDED FOR
gcc Check version: gcc -v	**5.4.0** Expected good[1]: gcc >= 4.8	Command-line Decomposition Methods[2]
Anaconda 3 Check version: conda --version Specific packages used: Python3, dask, pandas, numpy, matplotlib, pyyaml	**2019.10** Expected good: anaconda3 >= 2019.03	Python Tools & API[3]
ImageMagick Check version: convert --version	**6.8.9-9** Expected good: imagemagick >= 6.7.x	Python Tools & API

[1] Versions designated “Expected good” have not been tested, but do not have any known compatibility issues.
[2] perfCpAls, perfCpAlsNn, perfCpApr, perfCpAprStreaming, perfCpAprPdnr, perfCpAprPqnr, joinModeMaps.sh
[3] cvs2tensor.py, comp_top_k.py, query_decomp.py, visualize_decompsition.sh, basic-visualize_3.py, sync_labels.py, sync_stream.py, Python API

Prerequisite Installation Notes:

1. You could install necessary prerequisite packages with rpm, apt-get or yum

2. You could install Anaconda 3 for Python Tools & API from:
   https://docs.anaconda.com/anaconda/install/linux/

3. You may have to make the following change to provide permissions for ImageMagick post-installation:
   Change the line `<policy domain="coder" rights="none" pattern="PDF" />` in the file `/etc/ImageMagick-6/policy.xml` to `<policy domain="coder" rights="read|write" pattern="PDF" />`

2.3 Installation Process

After installing all prerequisites, ENSIGN itself can be installed with the included script `ENSIGN_install.sh`.

```
$> unzip ENSIGN-42.zip
$> cd ENSIGN-42
$> ./ENSIGN_install.sh
```

2.4 Testing the Installation

The installation can be tested with a single command. The package requires several environment variables to be set accordingly:

```
$> export ENSIGN_BASE=<path/to/ENSIGN-42>
$> export ANACONDA_HOME=<path/to/anaconda3>
$> export PATH=$ENSIGN_BASE/ENSIGN_4.2/bin:$PATH
$> export PATH=$ENSIGN_BASE/ENSIGN_4.2/Ensign-Py3/bin:$PATH
$> export PATH=$ANACONDA_HOME/bin:$PATH
$> export PYTHONPATH=$ENSIGN_BASE/ENSIGN_4.2/Ensign-Py3
$> export LD_LIBARY_PATH=$ENSIGN_BASE/ENSIGN_4.2/Ensign-CAPI/lib
```

To test the installation:

```
$> cd ENSIGN-42/ENSIGN_4.2/
$> ./ENSIGN.test_install
```

This command will run basic installation and runtime dependency checks, ENSIGN unit tests, and example tensor operations to determine if the installation is ready to use.

2.5 Support

Contact Reservoir Labs:

- Email: support@reservoir.com

3 Using the Command-line Decomposition Methods

This section provides an overview of the `Ensign-C` directory and how the executables built from inside and copied to `bin` can be used to run tensor decompositions. The `Ensign-C` directory looks like this (**bolded** items are directories):

Ensign-C
|---- **test_data**

3.1 Input Data

3.1.1 Tensor Input Format

1. Required *input tensor data* format:
    a. The first line should be "sptensor", indicating that the data is a sparse tensor
    b. The second line should be the number of modes
    c. The third line should be the size of each mode separated by spaces
    d. The fourth line is the number of nonzero entries
    e. The rest of the file has one row (line) per nonzero entry. Each row has multiple columns separated by tabs where the first column is the index in the first mode, the second column is the index in the second mode, etc. The last column is the value of the nonzero entry.
    f. Example:

```
sptensor
5
1481181 1899 1899 3 6067
2085108
0     0     1     0     0     1
1     0     2     0     2     1
2     2     3     1     0     1
3     3     1     2     1     1
...
```

2. Restrictions:
    a. Each mode is indexed from zero
    b. Each entry tuple (entry excluding value) must be unique[4]

3. Recommendations (for efficient computation):
    a. For each mode, all indices should be used in at least one nonzero entry

[4] Non-unique tensor entries are not caught as errors and result in undefined behavior. In this version of ENSIGN, duplicate entries are summed during tensor construction, but this behavior is not guaranteed to persist to future versions and duplicate entries should be avoided.

b. A tensor entry should not have a zero value

4. The `test_data` directory in `Ensign-C` contains various sub-directories, each of which holds test data for tensor decomposition. It also includes an example configuration file, `example_cfg.txt`. See Section 5.1 for more details on datasets included in the ENSIGN package and Section 3.2 for more details on configuration files.

3.1.2 Initial Guess Inputs

- Specifying Initial Guess Matrices

  For a mode-n tensor of size $I_0 \times I_1 \times \ldots \times I_{n-1}$, a user can optionally specify `n` input initial guess matrices (in `n` text files inside the directory specified in `<aux_dir>` argument). The `kth` initial guess matrix ($0 \leq k \leq n-1$) should be of size $I_k \times$ `<rank>` and is specified in a text file named `input<k>.txt` in the directory `<aux_dir>`. The input initial guess matrices are specified in the "input matrix format" described below. If initial guess matrices are not specified, the initial guesses are randomly generated.

  For example, for the Enron tensor (that is of size `105 x 105 x 27`), for a rank `20` decomposition, there are three text files `input0.txt`, `input1.txt`, and `input2.txt` in the directory `test_data/enron/input`. The matrices specified in the text files are, respectively, of the following sizes: `105 x 20`, `105 x 20`, and `27 x 20`.

- Required *matrix* format
  1. The first line should be "matrix", indicating that the data is a 2D matrix
  2. The second line should be the number of rows and columns
  3. The rest of the file has the matrix entries, one row per line and each column separated by spaces.
  4. Example:

```
matrix
1540  5
0.1   0.4   1.1   0.3   0.3
1.2   0.3   3.0   0.4   2.2
3.0   2.1   3.1   1.2   0.2
3.2   3.1   1.9   2.7   1.5
…
```

3.2 Configuration Files

Each of the command-line tools includes an auxiliary file directory option. This directory is used for specified initial guess matrices as explained in Section 3.1.2, however this directory may also contain a `cfg.txt` file that allows the user to specify parameters that are specific to the internal function of each decomposition application.

Valid `cfg.txt` file options are configured using key-value pairs to provide to the command-line tools. Key features of the configuration file format include:

- All text following a `#` symbol are ignored as comments
- All key-value pairs are written as `Key=Value` on single lines (no space between `Key`, `=`, and `Value`)
- For each method, a prefix (e.g., `als.*`) is optional. When the prefix is present, the key-value pairs are restricted to that class of algorithm. When the prefix is not present, the key-value pair applies to all algorithms (or ignored if not used). For example:
  - `cpals.MaximumIterations=150` applies only to CP-ALS
  - `MaximumIterations=150` applies to all valid methods with a maximum iterations configuration option
- When a value for MemoryLimit is not provided, the default limit is set to 80% of the system memory. The `cfg.txt` file can be used to set this higher if needed.
- An example ENSIGN configuration file `Ensign-C/test_data/example_cfg.txt` is provided below:

```
# Example ENSIGN Configuration File

# This file provides an example of key-value pairs for configuring ENSIGN
# command-line tools.

# Notes
# * No space between key/value pair (e.g., MemoryLimit=2147483648)
# * Works with values represented as scientific notation
# * Prefix (e.g., als.*) is optional - when present, restricts key to that
#   class of algorithm, when not present is applied to any algorithm
# * All comments and text after the '#' symbol are ignored

# All Algorithms
# If key is not set, default to 80% of total system memory
# MemoryLimit=2147483648 # 2 GB, max=none (in bytes)

# CP ALS
als.MaximumIterations=100 # max=10000
als.StopTolerance=1.0e-5 # max=1
```

```
# CP ALS STREAMING
alsstr.MaximumIterations=100 # max=10000
alsstr.StopTolerance=1.0e-5 # max=1

# CP ALS NN
alsnn.MaximumIterations=100 # max=10000
alsnn.StopTolerance=1.0e-5 # max=1

# CP APR
apr.MaximumOuterIterations=100 # max=10000
apr.MaximumInnerIterations=10 # max=10000
apr.StopTolerance=1.0e-4 # max=1

# CP APR STREAMING
aprstr.MaximumOuterIterations=100 # max=10000
aprstr.MaximumInnerIterations=10 # max=10000
aprstr.StopTolerance=1.0e-4 # max=1

# CP APR PDNR
pdnr.MaximumOuterIterations=100 # max=10000
pdnr.MaximumInnerIterations=10 # max=10000
pdnr.StopTolerance=1.0e-4 # max=1
pdnr.IsInexact=1 # 1 or 0

# CP APR PQNR
pqnr.MaximumOuterIterations=100 # max=10000
pqnr.MaximumInnerIterations=10 # max=10000
pqnr.StopTolerance=1.0e-4 # max=1
pqnr.IsInexact=1 # 1 or 0
pqnr.lbfgs_M=3 # max=10000
```

Each option is described in its respective method subsection of Section 3.4. If no `cfg.txt` is provided, or if an option is not assigned a value, the default option as listed is used.

3.3 Running with Multithreaded Execution

The decomposition methods use OpenMP to enable multithreaded execution. To run a multithreaded version of a tool from the command line, set the environment variable `OMP_NUM_THREADS` to the number of threads and then run the method. It is also recommended that you set `OMP_PROC_BIND` to true and/or set `GOMP_CPU_AFFINITY` to a binding appropriate to the platform.

For example:

- To run `perfCpAls` to decompose `enron` data into 10 components using the ENSIGN mode-specific data structure on 8 cores:

```
$> export OMP_NUM_THREADS=8
$> export OMP_PROC_BIND=true
$> export GOMP_CPU_AFFINITY="0-6:2 1-7:2"
$> perfCpAls test_data/enron/enron_data.txt 10 1
```

Setting `GOMP_CPU_AFFINITY` can have a significant impact on performance. For more information on the syntax for setting the variable, see:

https://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fCPU_005fAFFINITY.html

3.4 The Decomposition Methods

The `Ensign-C` directory encapsulates the routines for performing tensor decompositions. The examples below assume that the `PATH` variable is set correctly to include the created executables and that commands are executed from the `Ensign-C` subdirectory (where `test_data` is located).

The six (6) basic decomposition methods included in the ENSIGN package are:

METHOD	STATUS	PYTHON API[5]
perfCpAls	**Fully supported**	**Yes**
perfCpAlsNn	**Fully supported**	**Yes**
perfCpApr	**Fully supported**	**Yes**
perfCpAprStreaming	**Research use only**	No
perfCpAprPdnr	**Fully supported**	**Yes**
perfCpAprPqnr	**Fully supported**	**Yes**

The decomposition algorithms are described in detail below. A quick summary of options for each command can be accessed using the `--help` option.

[5] See Section 4 for details on accessing the decomposition methods through the Python API.

Each decomposition algorithm outputs operational metrics as part of its normal execution. These terms are defined in the Glossary in Section 6.

Fully Supported Algorithms

OUTPUT	ALGORITHM				
	perfCpAls	**perfCpAls Nn**	**perfCpApr**	**perfCpApr Pdnr**	**perfCpApr Pqnr**
Fit	Yes	Yes	Yes	Yes	Yes
Sparsity	Yes	Yes	Yes	Yes	Yes
Core sparsity	N/A	N/A	N/A	N/A	N/A
Norm scaling	Yes	Yes	Yes	Yes	Yes
Cosine similarity	Yes	Yes	Yes	Yes	Yes
KKT violation	N/A	N/A	Yes	Yes	Yes
Loglikelihood	N/A	N/A	Yes	Yes	Yes
Iteration count	Yes	Yes	Yes	Yes	Yes
Time	Yes	Yes	Yes	Yes	Yes

Research Use Only Streaming Algorithms

OUTPUT	ALGORITHM
	perfCpApr Streaming
Fit	Yes
Sparsity	Yes
Core sparsity	N/A
Norm scaling	Yes
Cosine similarity	Yes
KKT violation	Yes
Loglikelihood	Yes
Iteration count	Yes
Time	Yes

3.4.1 perfCpAls

Perform CANDECOMP/PARAFAC tensor decomposition using Alternating Least Squares method (CP-ALS) on sparse tensors.

Fully supported

```
USAGE: perfCpAls <tensor_file> <decomposition_rank>
<use_ENSIGN_opt=0|1> [OPTIONS]
```

- `<tensor_file>`: input tensor data file (with its path)
- `<decomposition_rank>`: rank of the CP decomposition
- `<use_ENSIGN_opt>`: 0/1
  - 1 signifies using Mode-Specific format
  - 0 signifies using Coordinate format

```
OPTIONS:
  -a <aux_dir>  auxiliary file directory
  -v            display verbose output
  -s <seed>     set random seed
  -i            write initial guess matrices to output directory
  -o <out_dir>  write output to specified directory
  --help        display all options and exit
```

Configuration file options: (optional prefix: als)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
`MemoryLimit`	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
`MaximumIterations`	The maximum number of iterations before the algorithm terminates	100	10000
`StopTolerance`	The minimum change in final fit per iteration before convergence	1.0e-5	1.0

Example: Run `perfCpAls` test to decompose Enron data into 20 components with "ENSIGN" functionality storing the output in `./results`

```
$> perfCpAls test_data/enron/enron_data.txt 20 1 -o ./results
```

References:

Kolda, T., Bader, B., *Tensor Decompositions and Applications*, SIAM Review, 51(3), pp. 455-500, 2009.

Baskaran, M., Meister, B., Vasilache, N., Lethin, R., *Efficient and Scalable Computations with Sparse Tensors*, In HPEC, September 2012.

Baskaran, M., Meister, B., Lethin, R., *Low-overhead Load-balanced Scheduling for Sparse Tensor Computations*, In HPEC, September 2014.

Baskaran, M., Henretty T., Pradelle, B., Langston, M.H., Bruns-Smith, D., Ezick, J., Lethin, R., *Memory-efficient Parallel Tensor Decompositions*, In HPEC, September 2017.

3.4.2 perfCpAlsNn

Perform CANDECOMP/PARAFAC tensor decomposition using Alternating Least Squares Nonnegative method (CP-ALS-NN) on sparse tensors

Fully supported

```
USAGE: perfCpAlsNn <tensor_file> <decomposition_rank>
<use_ENSIGN_opt=0|1> [OPTIONS]
```

- <tensor_file>: input tensor data file (with its path)
- <decomposition_rank>: rank of the CP decomposition
- <use_ENSIGN_opt>: 0/1
  - 1 signifies using Mode-Specific format
  - 0 signifies using Coordinate format

```
OPTIONS:
  -a <aux_dir>  auxiliary file directory
  -v            display verbose output
  -s <seed>     set random seed
  -i            write initial guess matrices to output directory
  -o <out_dir>  write output to specified directory
  --help        display all options and exit
```

Configuration file options: (optional prefix: alsnn)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
MemoryLimit	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
MaximumIterations	The maximum number of iterations before the algorithm terminates	100	10000
StopTolerance	The minimum change in final fit per iteration before convergence	1.0e-5	1.0

Example: Run `perfCpAlsNn` test to decompose Enron data into 20 components with "ENSIGN" functionality storing the output in `./results`

```
$> perfCpAlsNn test_data/enron/enron_data.txt 20 1 -o ./results
```

References:

Chen, D., Plemmons, R.J., *Nonnegativity Constraints in Numerical Analysis*, in Symposium on the Birth of Numerical Analysis, 2007.

Baskaran, M., Henretty T., Pradelle, B., Langston, M.H., Bruns-Smith, D., Ezick, J., Lethin, R., *Memory-efficient Parallel Tensor Decompositions*, In HPEC, September 2017.

3.4.3 perfCpApr

Perform CANDECOMP/PARAFAC tensor decomposition using Alternating Poisson Regression method (CP-APR), specifically using the Multiplicative Update (MU) algorithm, on sparse tensors.

Fully supported

```
USAGE: perfCpApr <tensor_file> <decomposition_rank>
<use_ENSIGN_opt=0|1> [OPTIONS]
```

- `<tensor_file>`: input tensor data file (with its path)
- `<decomposition_rank>`: rank of the CP decomposition
- `<use_ENSIGN_opt>`: 0/1
  - 1 signifies using Mode-Specific format
  - 0 signifies using Coordinate format

```
OPTIONS:
  -a <aux_dir>  auxiliary file directory
  -v            display verbose output
  -s <seed>     set random seed
  -i            write initial guess matrices to output directory
  -o <out_dir>  write output to specified directory
  --help        display all options and exit
```

Configuration file options: (optional prefix: apr)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
`MemoryLimit`	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
`MaximumOuterIterations`	The maximum number of iterations before the algorithm terminates	100	10000
`MaximumInnerIterations`	The number of iterations to perform within the per-iteration inner-loop	10	10000
`StopTolerance`	The minimum change in final fit per iteration before convergence	1.0e-4	1.0

Example: Run `perfCpApr` test to decompose Enron data into 20 components with "ENSIGN" functionality storing the output in `./results`

```
$> perfCpApr test_data/enron/enron_data.txt 20 1 -o ./results
```

References:

Chi, E., Kolda, T., *On Tensors, Sparsity, and Nonnegative Factorizations*, SIAM Journal on Matrix Analysis and Applications 33.4 (2012): 1272-1299.

Baskaran, M., Meister, B., Vasilache, N., Lethin, R., *Efficient and Scalable Computations with Sparse Tensors*, In HPEC, September 2012.

Baskaran, M., Meister, B., Lethin, R., *Low-overhead Load-balanced Scheduling for Sparse Tensor Computations*, In HPEC, September 2014.

Baskaran, M., Henretty T., Pradelle, B., Langston, M.H., Bruns-Smith, D., Ezick, J., Lethin, R., *Memory-efficient Parallel Tensor Decompositions*, In HPEC, September 2017.

3.4.4 perfCpAprStreaming

Perform CANDECOMP/PARAFAC APR tensor decomposition streaming update using a fast Multiplicative Update (MU) algorithm, on sparse tensor.

Research use only

```
USAGE: perfCpAprStreaming <path_to_existing_decomposition>
<update_tensor_file> <rank_update=(k>0)> <use_ENSIGN_opt=0|1>
[OPTIONS]
```

- <path_to_existing_decomposition>: location of a directory containing an existing and valid CP decomposition
- <update_tensor_file>: file containing updates to original data (with its path)
- <rank_update>: rank of the CP decomposition of update data
- <use_ENSIGN_opt>: 0/1
  - 1 signifies using Mode-Specific format
  - 0 signifies using Coordinate format

```
OPTIONS:
  -a <aux_dir>        auxiliary file directory
  -v                  display verbose output
  -s <seed>           set random seed
  -o <out_dir>        write output to specified directory
  -m <streaming_mode> index of streaming mode (default: 0)
  --help              display all options and exit

See Section 5.4 of the User Guide for additional usage information
```

Note that If the output directory is the same as the path to the existing decomposition, the original decomposition will be saved in a subdirectory of the existing decomposition path called base<i>, where i is the smallest nonnegative integer such that base<i> does not already exist.

Configuration file options: (optional prefix: aprstr)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
`MemoryLimit`	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
`MaximumOuterIterations`	The maximum number of iterations before the algorithm terminates	100	10000
`MaximumInnerIterations`	The number of iterations to perform within the per-iteration inner-loop	10	10000
`StopTolerance`	The minimum change in final fit per iteration before convergence	1.0e-4	1.0

Example: Run `perfCpAprStreaming` test to update the decomposition of the Enron data tensor from size 21 to size 27 along mode 2, decomposing the update data using a rank of 10, with "ENSIGN" functionality, and storing the output in `./results`.

```
$> perfCpAprStreaming test_data/enron/cp_apr_decomp/base \
test_data/enron/cp_apr_decomp/update/tensor_data.txt 10 1 \
-m 2 -o ./results
```

References:

Chi, E., Kolda, T., *On Tensors, Sparsity, and Nonnegative Factorizations*, SIAM Journal on Matrix Analysis and Applications 33.4 (2012): 1272-1299.

Letourneau, P., Baskaran, M., Henretty, T., Ezick, J., Lethin, R., *Computationally Efficient CP Tensor Decomposition Update Framework for Emerging Component Discovery in Streaming Data*, in HPEC, September 2018.

3.4.5 perfCpAprPdnr

Perform CANDECOMP/PARAFAC tensor decomposition using Alternating Poisson Regression method (CP-APR), specifically using the Projected Damped Newton Row subproblem (PDN-R) algorithm, on sparse tensors.

Fully supported

```
USAGE: perfCpAprPdnr <tensor_file> <decomposition_rank>
<use_ENSIGN_opt=0|1|2 (0=no-load-balancing-sorted, 1=load-balancing,
2=un-sorted-load-balancing)> [OPTIONS]
```

- `<tensor_file>`: input tensor data file (with its path)
- `<decomposition_rank>`: rank of the CP decomposition
- `<use_ENSIGN_opt>`: 0/1/2
  - 0 signifies using Mode-Specific format without processor load balancing
  - 1 signifies using Mode-Specific format with processor load balancing
  - 2 signifies using Mode-Specific format with unsorted processor load balancing

```
OPTIONS:
  -a <aux_dir>  auxiliary file directory
  -v            display verbose output
  -s <seed>     set random seed
  -i            write initial guess matrices to output directory
  -d            display load balancing debugging output
  -o <out_dir>  write output to specified directory
  --help        display all options and exit
```

Configuration file options (optional prefix: pdnr)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
`MemoryLimit`	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
`MaximumOuterIterations`	The maximum number of iterations before the algorithm terminates	100	10000
`MaximumInnerIterations`	The number of iterations to perform within the per-iteration inner-loop	10	10000
`StopTolerance`	The minimum change in final fit per iteration before convergence	1.0e-4	1.0
`IsInexact`	A Boolean that sets whether or not to relax the convergence criteria of the algorithm for faster decomposition. If set to true, does two inner iterations in the first outer iteration, and uses individual row subtolerances (defined as maximum of StopTolerance and (max KKT violation)/100) to determine convergence as opposed to total KKT violation over all matrices.	1	1

Example: Run `perfCpAprPdnr` test to decompose Enron data into 20 components with "ENSIGN" functionality storing the output in `./results`

```
$> perfCpAprPdnr test_data/enron/enron_data.txt 20 1 \
-o ./results
```

References:

Hansen, S., Plantenga, T., Kolda, T., *Newton-based Optimization for Kullback-Leibler Nonnegative Tensor Factorizations*, November 2014.

3.4.6 perfCpAprPqnr

Perform CANDECOMP/PARAFAC tensor decomposition using Alternating Poisson Regression method (CP-APR), specifically using the Projected Quasi-Newton Row subproblem (PQN-R) algorithm, on sparse tensors.

Fully supported

```
USAGE: perfCpAprPqnr <tensor_file> <decomposition_rank>
<use_ENSIGN_opt=0|1|2 (0=no-load-balancing-sorted, 1=load-balancing,
2=un-sorted-load-balancing)> [OPTIONS]
```

- `<tensor_file>`: input tensor data file (with its path)
- `<decomposition_rank>`: rank of the CP decomposition
- `<use_ENSIGN_opt>`: 0/1/2
  - 0 signifies using Mode-Specific format without processor load balancing
  - 1 signifies using Mode-Specific format with processor load balancing
  - 2 signifies using Mode-Specific format with unsorted processor load balancing

```
OPTIONS:
  -a <aux_dir>  auxiliary file directory
  -v            display verbose output
  -s <seed>     set random seed
  -i            write initial guess matrices to output directory
  -d            display load balancing debugging output
  -o <out_dir>  write output to specified directory
  --help        display all options and exit
```

Configuration file options: (optional prefix: pqnr)

OPTION NAME	DESCRIPTION	DEFAULT	MAXIMUM
`MemoryLimit`	The maximum number of bytes available for execution. A heuristic estimates the amount of memory required for a given input and stops if it exceeds this value.	80% of total system memory	None
`MaximumOuterIterations`	The maximum number of iterations before the algorithm terminates	100	10000
`MaximumInnerIterations`	The number of iterations to perform within the per-iteration inner-loop	10	10000
`StopTolerance`	The minimum change in final fit per iteration before convergence	1.0e-4	1.0
`IsInexact`	A Boolean that sets whether or not to relax the convergence criteria of the algorithm for faster decomposition. If set to true, does two inner iterations in the first outer iteration, and uses individual row subtolerances (defined as maximum of StopTolerance and (max KKT violation)/100) to determine convergence as opposed to total KKT violation over all matrices.	1	1
`lbfgs_M`	The number of previous iterates to keep in memory during L-BFGS	3	10000

Example: Run `perfCpAprPqnr` test to decompose Enron data into 20 components with "ENSIGN" functionality storing the output in `./results`

```
$> perfCpAprPqnr test_data/enron/enron_data.txt 20 1 \
-o ./results
```

References:

Hansen, S., Plantenga, T., Kolda, T., *Newton-based Optimization for Kullback-Leibler Nonnegative Tensor Factorizations*, November 2014.

3.5 Decomposition Results

3.5.1 CP Decomposition Results Format

1. Each CP tensor decomposition (those performed by **perfCpAls**, **perfCpAlsNn**, **perfCpApr**, **perfCpAprPdnr**, **PerfCpAprPqnr**) produces a `weights.txt` file and then one file for each mode i, `decomp_mode_<i>.txt`. To support streaming decompositions, each CP tensor decomposition also produces a `streaming.txt` file. By default these files are produced in the current directory. If the user specifies an output directory with the `-o` flag, then the files will be written to the specified directory.

2. The `weights.txt` file contain a simple list of numbers, one on each line. Each number is a weight, lambda, of a component that is a measure of how much of the total variance that particular component captures relative to other components. The weight on line 0 is the weight of the first component and the weight on line R-1 is the weight of the last component.

3. The `decomp_mode_<i>.txt` file contains the factor matrix for the ith mode. These files use the matrix file format described in section 3.1.2.

4. The streaming.txt file contains fit information needed if the decomposition is used as the base of a streaming decomposition. The format of the file is as follows:
    a. The first line is “#StreamingDatalessFit”
    b. The second line is the norm of the tensor
    c. The third line is the residual error between the decomposition and tensor
    d. The fourth line is the number of factors (rank of the decomposition)
    e. The rest of the file contains the inner product between the tensor and each rank-1 component of the decomposition (one line per component)
    f. Example:

```
#StreamingDatalessFit
6123.29764097745
4152.469503172811
5
18170488.00301399
19511.74262721192
14453.44862483238
1412513.288619133
10168.18539017305
```

5. In addition to the output files, the CP decompositions report statistics in the console on the number of iterations performed, convergence metrics, and quality of solution (e.g., fit, loglikelihood, KKT Violation, sparsity). These terms are defined in the Glossary in Section 6.

3.5.2 Analyzing and Visualizing CP Results

1. If the Python command-line tools & API were installed, then included inside the `Ensign-Py/bin` subdirectory is a shell script to visualize the results of the CP and Tucker tensor decompositions. The script takes as parameters the location of the directory containing the results of the decomposition and a target output file and produces a PDF that visualizes the results.

2. How to use the script:
   The name of the visualization script is `visualize_decomposition.sh`. After running one of the decompositions, a `decomp_mode_<i>.txt` result file will be generated for each mode of the tensor as well as a `weights.txt` file. By default, these files will be generated in the current directory.

For CP decompositions only, if files `map_mode_<i>.txt` are available, they will be used to generate labels for the top scoring elements in each mode.

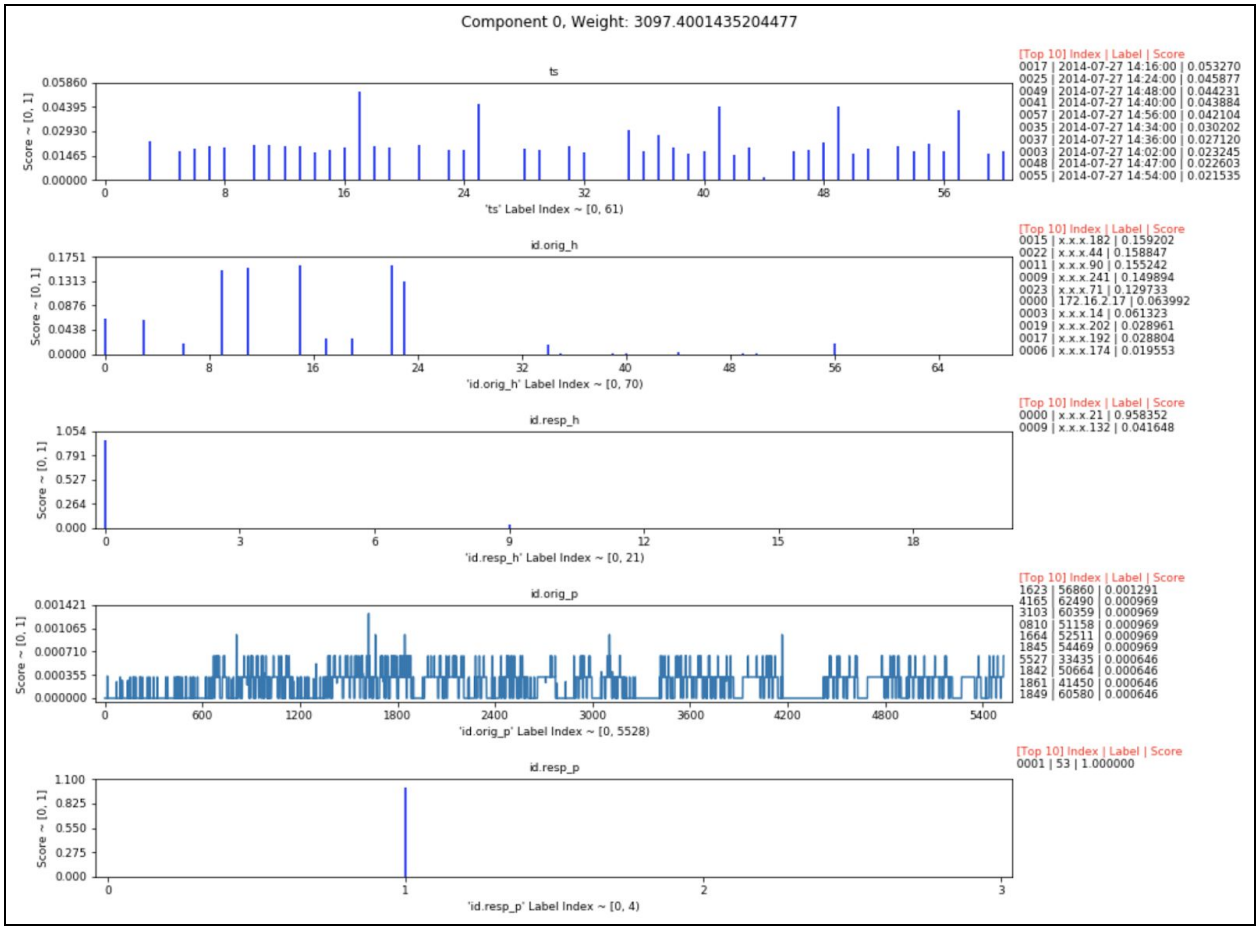
To run the visualization script, enter:

```
$> visualize_decomposition.sh [OPTIONS] \
</path/to/results_directory> [output_file.pdf]
```

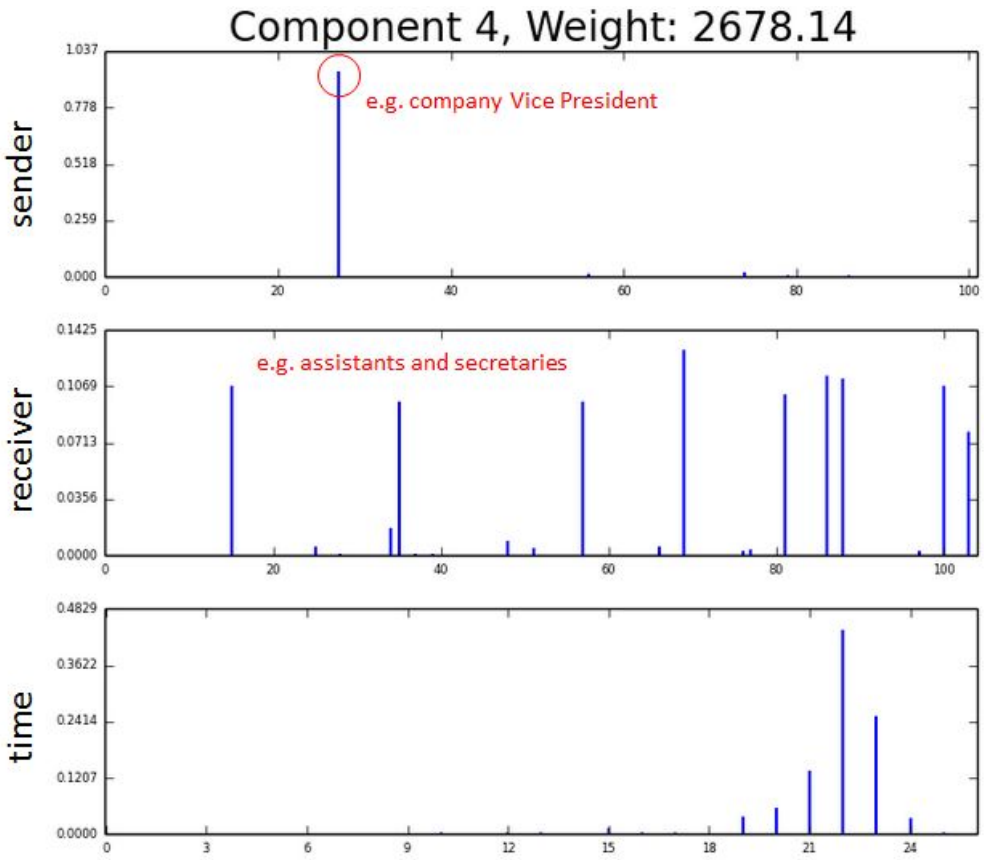
After running the script, a PDF titled *output_file.pdf* (default: *decomp.pdf)* will be created in the specified location. This PDF will contain images of each component of the decomposition (one per page) sorted in descending order by weight (or sum of weights in the joint case).

Usage instructions and examples for `visualize_decomposition.sh` are provided in Section 4.2.6.

3. How is the data visualized:
    - Each image features one plot per mode. So if the modes are *sender*, *receiver*, and *time*, the first plot will represent senders in this component, the second will represent receivers and the third, time. The x-axes represent the indices in each mode. So if there are 100 senders in the tensor, the x-axis of the sender plot would go from 1-100, representing the corresponding senders. The y-axes are the scores (i.e., the actual values produced by the decomposition).



In this example we see a visualization of a decomposition performed on DNS logs. The information on the right-hand side shows the labels associated with the highest scoring indices of each mode. In the last mode (destination port), we can see that the only non-zero index is port 53. Likewise, we can see that the destination IP mode is dominated by the IP ‘x.x.x.21’.



- The score of a particular index represents how strongly that index is related to other information in the component. For example, if in component 0, the index of the highest sender score is Vice President Smith and the highest receiver scores are spread across multiple secretaries and assistants, then component 0 represents the activity between V.P. Smith and his administrative assistants at the times that also have high scores.

4 Using the Python Command-line Tools & API

4.1 Introduction

ENSIGN includes a suite of Python 3 tools and Python 3 compatible API, Ensign-Py. The tools assist with building sparse tensors and examining the output of the tensor decompositions. In addition to providing access to the tools, the API also includes support for manipulating sparse tensors and performing and manipulating the output of static CP decompositions. Ensign-Py uses the NumPy `ndarray` class and Pandas DataFrames to represent numerous aspects of sparse tensors and decomposition results. Because of this, the large ecosystem of existing Python data science tools can be used for analysis of tensors and decompositions.

Ensign-Py is installed by default in the `<ENSIGN_4.2_dir>/Ensign-Py3` directory. In order to access Ensign-Py from your Python program this directory must be added to the `PYTHONPATH` environment variable. Further, the directory `<ENSIGN_4.2_dir>/Ensign-CAPI/lib` must be added to the `LD_LIBRARY_PATH` environment variable in order to use Ensign-Py.

An overview of the ENSIGN Python command-line tool suite is provided in Section 4.2. Ensign-Py includes full HTML API documentation, but basic usage is summarized in Section 4.3. Ensign-Py includes a demonstration application, described in Section 4.4. Finally, Section 4.5 provides pointers to Python data science tools that can be used for further analysis of Ensign-Py data.

4.2 Command-line Tool Usage

To use the Python command-line tools, ensure that the following environment variables are set correctly:

- `PATH` must contain `<ENSIGN_4.2_dir>/Ensign-Py3/bin`
- `PYTHONPATH` must contain `<ENSIGN_4.2_dir>/Ensign-Py3`
- `LD_LIBRARY_PATH` must contain `<ENSIGN_4.2_dir>/Ensign-CAPI/lib`

4.2.1 csv2tensor.py

Builds a sparse tensor from a given CSV file or Bro log.

```
usage: csv2tensor.py [-h] [-b <bintype1>,<bintype2>, ...]
                     [-c <col1>,<col2>, ...] [-d DELIMITER]
                     [-e ENTRIES] [-f <col1>,<col2>, ...] [-l] [-s]
                     [-t <type1>,<type2>, ...]
                     <csv_file> <tensor_directory>

positional arguments:
  <csv_file>              CSV input file. Required.
  <tensor_directory>      Tensor output directory. Required.

optional arguments:
  -h, --help
            Show this help message and exit.
  -b <bintype1>,<bintype2>, ..., --binning <bintype1>,<bintype2>, ...
            Binning method to use for each column. Optional. Legal
            values are 'binsize=<float64>', ip4subnet=<mask>',
            'log10', 'round=<int64>', and 'none'. Columns containing
            times can be binned by 'second', 'minute', 'hour', 'day',
            'month', and 'year'. Default: <all 'none'>
  -c <col1>,<col2>, ..., --columns <col1>,<col2>, ...
            Columns to use as tensor modes. Optional. Default: <all
            columns>
  -d DELIMITER, --delimiter DELIMITER
            Delimiter to use when fusing columns. Optional.
            Default: ' '
  -e ENTRIES, --entries ENTRIES
            Tensor entry calculation method. Optional. Legal values
            are 'count' and 'boolean'. Default: 'count'
  -f <col1>,<col2>, ..., --fuse-columns <col1>,<col2>, ...
            Columns to fuse into a single column. Optional. Default:
            no columns are fused
  -l, --log
            Treat input as a Bro log. Optional. Default: Treat input
            as a CSV
  -s, --sort
            Sort string values when mapping to indices. Optional.
```

```
Default: String values are not sorted
```

```
-t <type1>,<type2>, ..., --types <type1>,<type2>, ...
        Type of each column. Optional. Legal values are 'str',
        'int64', 'float64', 'ip', and 'time=<format>'. <format>
        is a strftime-compatible description of time in file
        e.g., '%Y-%m-%d %H:%M:%S'. Additionally, epoch timestamps
        can be specified using the format string '%E'. Default:
        <all 'str'>
```

Example 1: Create a sparse tensor with default options using a CSV file.

```
$> csv2tensor.py </path/to/csv> </path/to/output/dir/>
```

This will output a single `tensor_data.txt` file and a `map_mode_<i>.txt` file for each column `<i>` in the CSV file.

Example 2: Use only a specific subset of the original features as the modes of the new tensor with the `-c` flag:

```
$> csv2tensor.py -c col1,col2,col3 \
</path/to/csv> </path/to/output/dir/>
```

Only three `map_mode_<i>.txt` files will be created as per the list passed to the `-c` flag.

Example 3: Create a boolean sparse tensor from a Bro log with timestamp, source IP, and destination IP. Bin the timestamp by hour and source IP by subnet.

```
$> csv2tensor.py -c ts,id.orig_h,id.resp_h \
            -b hour,ip4subnet=255.255.0.0,none \
            -t time=%E,ip,string \
            -e boolean -l \
            </path/to/bro_log> </path/to/output/dir/>
```

Again, only a few columns will be used to create the tensor, however the source file used to build the tensor is a Bro log. The `-b` flag has the timestamp binned by the hour and the source IP binned by subnet. The `-t` flag indicates that the ts column is represented as an epoch timestamp, the id.orig_h column as an IP address, and the id.resp_h column as a string. The entries of the tensor will be boolean values.

4.2.2 comp_top_k.py

Print the top k scoring labels for each mode of specified components in a decomposition.

```
usage: comp_top_k.py [-h] [-k TOP_K] <decomp_dir> <comp_ids>

positional arguments:
  <decomp_dir>            Path to directory containing
                          decomposition. Required.
  <comp_ids>              Component IDs in comma-separated inclusive
                          intervals. e.g. 0-3,5,7-10 returns top k
                          for components 0,1,2,3,5,7,8,9,10.
                          Required.

optional arguments:
  -h, --help
            Show this help message and exit.
  -k TOP_K, --top-k TOP_K
            Print the top k labels for each mode of the specified
            components as scored by their associated entries in the
            factor matrices. Optional. Default: 10
```

Example: Print out the top three scoring labels for each mode of components 0, 1, and 2 of the decomposition residing in `</path/to/decomp>`.

```
$> comp_top_k.py -k 3 </path/to/decomp> 0-2
```

4.2.3 query_decomp.py

Print all components containing the query label in the given mode(s). Up to *k* components are printed where `<query_label>` has a non-zero score.

```
usage: query_decomp.py [-h] [-k TOP_K]
                       <query_label> <decomp_dir>:<modes>
                       [<decomp_dir>:<modes> ...]

positional arguments:
  <query_label>           Query label. Required.
  <decomp_dir>:<modes>    Key-value pairs for decomposition
                          directories and corresponding modes. There
                          can be one or more pairs. Example:
                          ~/decomp_dir/location:1,3.

optional arguments:
  -h, --help
          Show this help message and exit.
  -k TOP_K, --top-k TOP_K
          Return the top k components where <query_label> has the
          highest score. Optional. Default: 10
```

Example 1: Print out all the components and scores associated containing the label `x.x.x.21` in modes 1 and 2 of the decomposition in `</path/to/decomp>`.

```
$> query_decomp.py x.x.x.21 </path/to/decomp>:1,2
```

Example 2: Print out the components containing the label `x.x.x.21` in modes 1 and 2 of the decomposition in `</path/to/decomp1>` and modes 2 and 3 of the decomposition in `</path/to/decomp2>` that have at least the third highest score associated with the query label in each mode.

```
$> query_decomp.py -k 3 x.x.x.21 \
     </path/to/decomp1>:1,2 </path/to/decomp2>:2,3
```

4.2.4 sync_labels.py

Synchronize specified modes between a set of tensor directories with or without decompositions. Allows for an arbitrary number of tensors and/or decompositions to be synchronized to use the same label-index maps between sparse tensors, factor matrices, and mode maps.

```
usage: sync_labels.py [-h] [-m MODES_TO_SYNC] [-o OUTPUT] [-i] [-v]
<dirs>

positional arguments:
  <dirs>                  Paths to tensors/decomposition directories
                          to synchronize. Must be in a comma
                          separated list. e.g.
                          'path/to/tensor1,path/to/tensor2'.
                          Required.

optional arguments:
  -h, --help
            Show this help message and exit.
  -m MODES_TO_SYNC, --modes-to-sync MODES_TO_SYNC
            Mapping of modes for synchronization, expects a
            comma-separated list of modes for each tensor separated
            by colons. Optional. Default: map all corresponding
            modes. Example: 0,1:1,2 synchronizes mode 0 of tensor A
            to mode 1 of tensor B. Similarly, mode 1 of tensor A and
            mode 2 of tensor B will be synchronized.
  -o OUTPUT, --output OUTPUT
            Directory to save the synchronized base and update to.
            Directories named 'base' and 'update' in the output
            directory will be overwritten. Default: Save to current
            working directory in a folder named 'output'.
  -i, --in-place
            Overwrite the specified base_dir and update_dir. If
            toggled, the output option will be ignored. Default:
            False
  -v, --verbose
            Display verbose output. Default: False
```

Example 1: Synchronize all of the labels between a tensor in `</path/to/tensor>` and a decomposition in `</path/to/decomp>` and save the synchronized versions in a directory `<synced_dir>`.

```
$> sync_labels.py -o <synced_dir> \
</path/to/tensor>,</path/to/decomp>
```

Example 2: Synchronize the labels in modes 1 and 2 of a tensor in `</path/to/tensor>` to the labels of modes 2 and 3 of a decomposition in `</path/to/decomp>` and save the synchronized versions in their original directories, overwriting (`-i` flag) the original versions.

```
$> sync_labels.py -i -m 1,2:2,3 \
</path/to/tensor>,</path/to/decomp>
```

4.2.5 sync_stream.py

Synchronize specified modes between a base tensor/decomposition and an update tensor/decomposition while preserving ordering of the base tensor's labels. For use with streaming decompositions.

```
usage: sync_stream.py [-h] [-m STREAMING_MODE] [-o OUTPUT] [-i] [-p]
                      [-v] <base_dir> <update_dir>

positional arguments:
  <base_dir>              Path to base tensor/decomposition
                          directory. Required.
  <update_dir>            Path to update tensor/decomposition
                          directory. Mode maps, tensor data, and
                          decomposition files are rewritten in terms
                          of the new label indices. Required.

optional arguments:
  -h, --help
            Show this help message and exit.
  -m STREAMING_MODE, --streaming-mode STREAMING_MODE
            Mode to stream along. Every other mode will be
            synchronized. Overlapping labels will be removed from the
            update and be logged in a file named: 'dropped.csv'.
  -o OUTPUT, --output OUTPUT
            Directory to save the synchronized base and update to.
            Directories named 'base' and 'update' in the output
            directory will be overwritten. Default: Save to current
            working directory in a folder named 'output'.
  -i, --in-place
            Overwrite the specified base_dir and update_dir. If
            toggled, the output option will be ignored. Default:
            False
  -p, --ignore-base-tensor
            Performance option. Will only synchronize the base
            decomposition to the update tensor/decomposition. Will
            NOT synchronize the base tensor. Default: False
  -v, --verbose
            Display verbose output. Default: False
```

Note on `dropped.csv`**:**
Overlapping labels in the streaming mode are removed from the update tensor. As a result, any tensor entries in the update tensor containing those labels have to be removed as well. These entries are logged to a file in the update output directory named `dropped.csv`. This has a similar format to the `tensor_data.txt` files except without a header, the labels themselves are displayed instead of their indices, and the file is in CSV format.

Example: Prepare a base decomposition and update tensor for a mode-0 streaming decomposition. Synchronize along modes 1,2,3 for both the base and update. Clean the overlapping labels of mode 0 from the update tensor and save the synchronized output in a directory called `<synced_dir>`.

```
$> sync_stream.py -m 0 -o <synced_dir> \
</path/to/base/decomp> </path/to/update/tensor>
```

How does `sync_labels.py` **relate to** `sync_stream.py`**?**

Both tools achieve the similar goal of synchronizing the contents of tensor directories with respect to the label maps.

`sync_labels.py` is a more general-purpose synchronization tool. The key differences are that it can synchronize an arbitrary number of tensors/decompositions and that it does **not** preserve the ordering of labels on any of the tensors/decompositions.

`sync_stream.py` is a tool designed specifically for streaming decompositions. It will maintain the original ordering of labels of the base decomposition and additional options are provided to customize functionality.

See Section 5.4 for more information about synchronization and the motivation behind `sync_stream.py`.

4.2.6 visualize_decomposition.sh

Generate a static PDF component visualization of a CP or Tucker decomposition

```
usage: visualize_decomposition.sh [-c COMPONENTS] [-a] [-b]
                                  <input_dir> <output_file>

positional arguments:
  <input_dir>             Path to directory containing the
                          decomposition to visualize. Required.
  <output_file>           File path specifying where the output file
                          will be saved. Optional. Default:
                          'decomp.pdf'

optional arguments:
  --help
            Show this help message and exit.
  -c COMPONENTS
            Number of components to include in visualization (the top
            k values from the core tensor will be used). Default 20
            for Tucker, or all components for CP.
  -a (abs-core-entries)
            For Tucker decompositions, consider the absolute values
            of core tensor entries when computing top k values.
  -b (basic-viz)
            For CP decompositions, use the unlabelled visualization.
            (Simpler to view and faster to generate).
```

Example: Generate visualizations of the top five (5) components of a decomposition located in `./results/` and output the PDF to a file called `viz.pdf`.

```
$> visualize_decomposition.sh -c 5 ./results/ viz.pdf
```

4.3 API Usage

Full HTML documentation of the Ensign-Py API is included in the `<ENSIGN_4.2_dir>/Ensign-Py3/ENSIGN_Python_API_Docs/index.html` file. Open this file in your web browser of choice to view detailed descriptions of classes and functions in Ensign-Py.

The following sections show common operations in Ensign-Py. To work with the code itself, a similar walkthrough is provided in the Jupyter notebook `<ENSIGN_4.2_dir>/Ensign-Py3/demo/Tour_of_Ensign-Py.ipynb`

4.3.1 Import Ensign-Py Modules

Ensign-Py modules are imported with standard Python syntax. In any Python program, the following Python code will import the core components of Ensign-Py.

```
import ensign.sptensor as spt
import ensign.cp_decomp as cpd
import ensign.csv2tensor as c2t
from ensign.comp_top_k import get_topk
from ensign.query_decomp import query_decomp
from ensign.synchronize_labels import synchronize_labels
from ensign.synchronize_labels import synchronize_labels_stream
import ensign.visualize as viz
```

Throughout our examples we import the `sptensor` module as `spt`, the `cp_decomp` module as `cpd`, and the `csv2tensor` module as `c2t`.

4.3.2 Build a Sparse Tensor

The `csv2tensor` module (imported here as `c2t`) allows users to generate a sparse tensor from delimiter separated value files (e.g., .csv files) or Bro logs.

Start by loading the CSV with Pandas:

```
import pandas as pd
test_data = \
'<ENSIGN_4.2_dir>/ENSIGN-42/ENSIGN_4.2/ENSIGN_GUI/test_data/'
df = pd.read_csv(test_data + 'dns_example.csv')
```

Example 1: Create a sparse tensor with default options using a CSV file.

```
sp_tensor = c2t.df2tensor(df)
```

Example 2: Use only a specific subset of the original features as the dimensions of the new tensor with the columns argument.

```
sp_tensor = c2t.df2tensor(df, ['col1', 'col2', 'col3'])
```

Example 3: Create a sparse tensor with columns including timestamp, origin IPs and ports, and destination IPs and ports. Bin the timestamp by the minute.

```
sp_tensor = c2t.df2tensor( \
      data, \
      columns=['ts', 'id.orig_h', 'id.resp_h', 'id.orig_p', \
      'id.resp_p'], \
      types=['time=%E', 'str', 'str', 'str', 'str'], \
      binning=['minute', 'none', 'none', 'none', 'none'] \
)
```

The df2tensor function returns an `SPTensor` object. More information about the `SPTensor` class can be found in the docs.

4.3.3 Load a Sparse Tensor

After importing Ensign-Py sptensor module, a sparse tensor can be loaded with the following Python code:

```
t = spt.read_sptensor('path/to/tensor/dir)
```

This code loads the sparse tensor file located in `path/to/tensor/dir` into the variable `t`. A description of the sparse tensor file format can be found in Section 3.1.1.

To save a sparse tensor, simply use the following method:

```
t.write('path/to/save/directory')
```

4.3.4 Decompose a Sparse Tensor

After loading a sparse tensor `t`, it can be decomposed using with the CP-ALS algorithm:

```
rank = 3
decomp_als = cpd.cp_als(t, rank, 'path/to/als_decomp')
```

The code above sets variable `rank = 3` and uses this for a CP ALS decomposition of tensor t. The output of the decomposition is stored in the variable `decomp_als` and also written to the filesystem in the directory `path/to/als_decomp`. The output directory argument can be omitted to avoid IO and keep the decomposition in memory.

Similar code for CP-APR and CP-APR-PDNR decompositions is shown below

```
decomp_apr = cpd.cp_apr(t, rank, 'path/to/apr_decomp_output')
decomp_pdnr = cpd.cp_apr_pdnr(t, rank, \
                              'path/to/pdnr_decomp_output')
```

The tensor, and rank arguments to decomposition functions are required. See the Ensign-Py API documentation in the `Ensign-Py3/ENSIGN_Python_API_Docs` directory for optional parameters that enable fine grain control over decomposition behavior.

4.3.5 Access Decomposition Results

We can access the results of a decomposition through the `CPDecomp` class in the `cp_decomp` module. In the following code, the variable `decomp` is the result of a successful decomposition. The results of the decomposition are stored in the variables `weights` and `factors`.

```
weights = decomp.weights; factors = decomp.factors
```

The `weights` variable is a 1D `ndarray` of length `rank` and contains the weight of each decomposition component. The `factors` variable is a list of `ndarray` where each entry is a `ndarray` with `shape = (mode_size, rank)`.

4.3.6 Plot a Decomposition

An entire decomposition can be plotted and displayed inline with a Jupyter notebook or saved to disk with the `plot_component()` function. The following will display the blue-line chart inline:

```
viz.plot_component(decomp, component_id)
```

To save to disk, specify the third argument `inline` as `False`:

```
viz.plot_component(decomp, component_id, False)
```

This will save the output PNG file in the current working directory as `comp_<comp_id>.png`.

4.3.7 Save and Reload a Decomposition

A decomposition can be saved to the filesystem using the `write_cp_decomp_dir()` function.

```
cpd.write_cp_decomp_dir('path/to/save/dir', decomp)
```

Decompositions can also be loaded from the filesystem using the `read_cp_decomp_dir()` function.

```
decomp = cpd.read_cp_decomp_dir('path/to/save/dir')
```

Fine-grained control of reading decompositions is provided by optional parameters. See the Ensign-Py API documentation in the `Ensign-Py/ENSIGN_Python_API_Docs` directory for further details.

4.3.8 Reconstruct a Decomposition into a Tensor

A decomposition can be reconstructed "into" the tensor it was created from. This is accomplished using the `reconstruct_into()` function.

```
tensor = cpd.reconstruct_into(decomp, tensor)
```

A sum-of-outer-products is computed for each nonzero entry of the original tensor. In other words, only those elements that are nonzero in the original tensor are reconstructed.

Reconstructions can also be performed from a subset of the Components in a decomposition. In the example below, only Components 1 and 3 are reconstructed. The list of Components to reconstruct is passed in the `comp_ids` parameter.

```
partial_tensor = cpd.reconstruct_into(decomp, tensor, \
                                      comp_ids=[1,3])
```

See the Ensign-Py API documentation in the `Ensign-Py/ENSIGN_Python_API_Docs` directory for a complete description of all available parameters to the `reconstruct_into()` function.

4.3.9 Calculate Per-Entry Fit of a Decomposition

The degree to which a reconstructed decomposition fits the original tensor[6] can be computed on a per-entry basis. This is accomplished using the `get_fit_per_entry()` function.

```
fits = cpd.get_fit_per_entry(decomp, tensor)
```

Fit is calculated for each element of the tensor and the result is returned as a two-dimensional `ndarray` with `dtype=float64`. Each row of the array represents one nonzero entry fit value. The first *m* columns of the array contain the index of the entry in each mode and the last column contains the fit value. This is the same format used in the `entries` attribute of the `tensor` argument passed to the function.

A subset of the highest or lowest fit values can be returned by passing a value for the `top_k` parameter to the function. Positive values return the `top_k` smallest fit entries while negative values return the `top_k` largest. The below example produces the 10 elements with the lowest (worst) fit when reconstructing `decomp` into `tensor`.

```
top_k_fits = cpd.get_fit_per_entry(decomp, tensor, top_k=10)
```

Low fits can be indicative of anomalous entries in the original tensor. See the Ensign-Py API documentation in the `Ensign-Py/ENSIGN_Python_API_Docs` directory for further details.

4.3.10 Comp-Top-K & Query-Decomp

Two command-line tools `comp_top_k.py` and `query_decomp.py` are included with Ensign-Py to help analyze decomposition results. `comp_top_k.py` allows the user to find the most significant indices in selected components of a decomposition.

For example:

```
decomp = cpd.read_cp_decomp_dir('path/to/decomp/')
topk = get_topk(decomp.factors, decomp.labels, [1,2,3], 5)
```

This returns tuples containing the labels, mode-index, and score associated with the top 5 labels from every mode of components 1, 2, and 3.

Query-Decomp allows users to search a decomposition for non-zero scored instances of a particular label.

[6] Fit is calculated as 1 - | (*t* - *r*) / *t* | where *t* is the original tensor entry value and *r* is the reconstructed tensor entry value. A fit value of 1 indicates an exact reconstruction of the entry. Fit values less than 1 indicate an inexact reconstruction. Fit values can be negative, and a smaller fit value corresponds to a more inaccurate reconstruction. In other words, a fit of -1 is worse than a fit of 0.3.

For example:

```
decomp = cpd.read_cp_decomp_dir('path/to/taxi/decomp')
results = query_decomp(decomp.factors, decomp.labels, \
[1,2], 24)
```

The results are returned as a Pandas dataframe with columns 'Score', 'Mode', and 'Component'. Each row represents a factor matrix entry where the query-label has a non-zero score.

4.3.11 Synchronize Labels

Occasionally, particular use cases require a set of tensors or decompositions to share common mode maps. To retroactively 'synchronize' tensors and/or decompositions, `synchronize_labels.py` can be used. This module provides two functions; `synchronize_labels` and `synchronize_labels_stream`.

For a general case where an arbitrary number of tensors/decompositions need to be synchronized, `synchronize_labels` may be used.

This example synchronizes every mode of the tensors and returns the synchronized versions in memory. *Note:* A mixture of tensors and decompositions can be passed to one call of `synchronize_labels`.

```
# Paths to existing tensor directories
tensors = ['path/to/tensor1', 'tensor2', 'tensor3']
synced_tensors = synchronize_labels(tensors)
```

Synchronizing labels is particularly useful for preparing a base decomposition and updating a tensor for a streaming decomposition. In that case, all the modes must be synchronized besides the streaming mode. `synchronize_labels_stream` is made for just this case. It preserves the ordering of the labels in the base decomposition. Additionally, it removes the labels from the update tensor in the streaming mode that also exist in the streaming mode of the base.

The following is a quintessential example of synchronizing a decomposition and tensor of order 3, and preparing them for streaming along mode 0.

```
base = cpd.read_cp_decomp_dir('path/to/base/decomp')
update = spt.read_sptensor('path/to/update/tensor')
synced_base, synced_update = synchronize_labels_stream( \
      base, \
      update, \
      streaming_mode=0)
```

4.4 Demonstration Application

Ensign-Py includes a demonstration application that provides examples of basic sparse tensor decomposition operations. The basic demo can be found in the `<ENSIGN_4.2_dir>/Ensign-Py/demo/basic` directory.

To build and run the demonstration application:

```
$> cd <ENSIGN_4.2_dir>/Ensign-Py3/demo/basic
$> make
```

Additionally, a Jupyter Notebook showcasing a standard workflow in Ensign-Py can be found at `<ENSIGN_4.2_dir>/Ensign-Py/demo/Tour\ of\ Ensign-Py.ipynb`

To view the notebook locally, run the following:

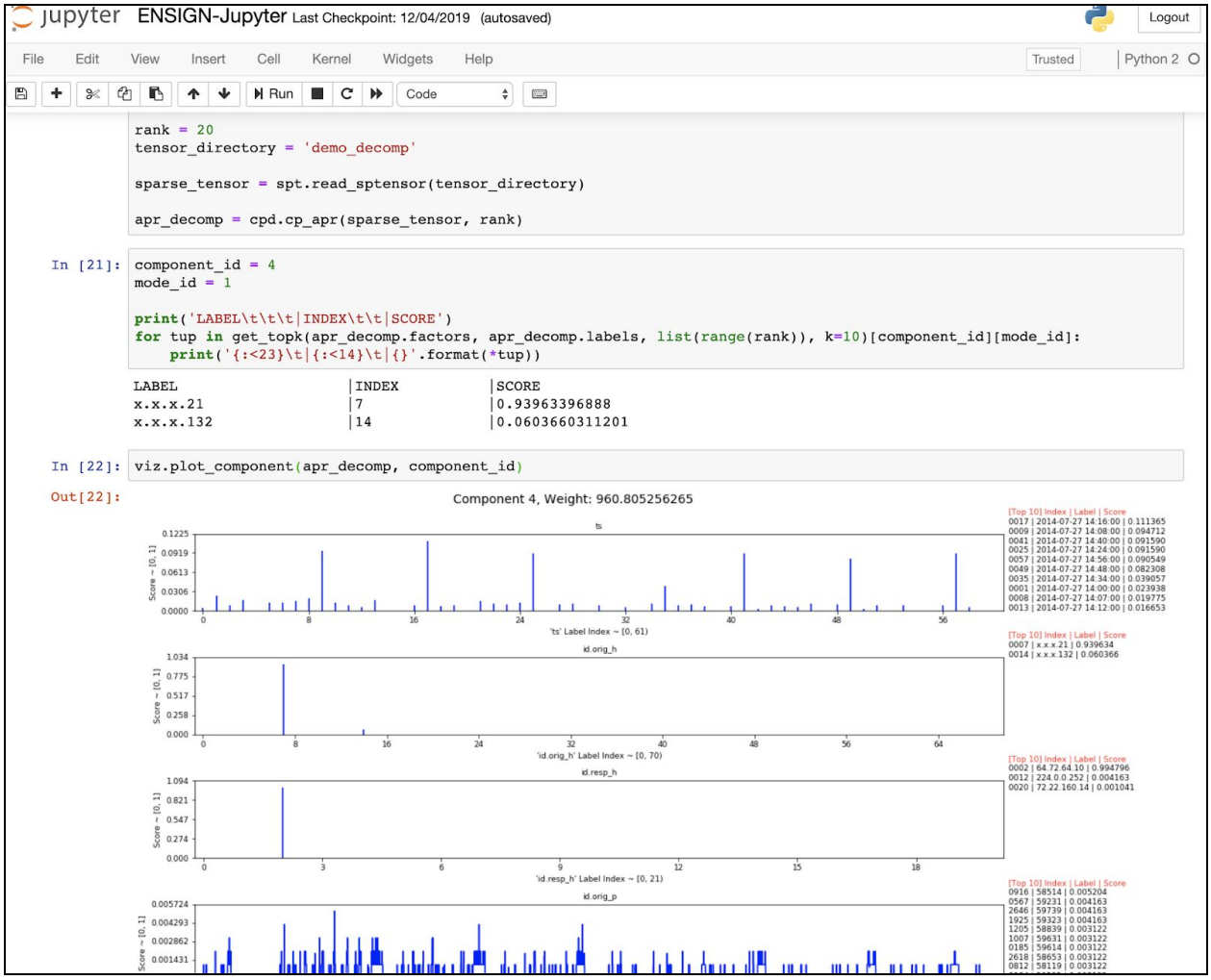
```
$> cd <ENSIGN_4.2_dir>/Ensign-Py3/demo
$> jupyter notebook
```

In the browser window that opens, select `Tour of Ensign-Py.ipynb`.

4.5 Other Analysis Tools

Ensign-Py, through its use of the NumPy `ndarray` for key data structures, can be integrated with a large number of Python data science tools and libraries. We list some of these below.

Jupyter Notebook (http://jupyter.org): An interactive environment for interactive development of Python code.



Anaconda (https://www.anaconda.com): An open-source distribution of Python that eases package management and deployment.

NumPy (http://www.numpy.org): A library that provides foundational numerical computing functions and data structures.

SciPy (https://www.scipy.org): A library that provides high level math, science, and engineering functions.

Pandas (http://pandas.pydata.org): A library that provides foundational data analysis functions and data structures.

scikit-learn (http://scikit-learn.org): A library that provides foundational machine learning algorithms.

Matplotlib (https://matplotlib.org): A library that provides foundational 2D plotting capabilities.

Seaborn (https://seaborn.pydata.org): A library that provides high level 2D plotting capabilities.

Bokeh (http://bokeh.pydata.org): A library that provides foundational interactive plotting capabilities.

StatsModels (http://www.statsmodels.org): A library that provides foundational statistical modelling functions and data structures.

5 Additional Information

The information in this section provides additional background intended to assist with becoming familiar with ENSIGN and understanding use cases for some of the more specialized decomposition routines.

5.1 Notes on Included Test Cases

The ENSIGN includes several sample inputs. These inputs are provided as use cases for demonstrating functionality and are not intended to be used as performance benchmarks.

5.1.1 Ensign-C/test_data

1. **`cyber`**: This tensor was generated from Reservoir's internal network traffic messages.
2. **`enron`**: This tensor was created from "The Enron Email Dataset - Database Schema and Brief Statistical Report," by Jitesh Shetty and Jafar Adibi.
3. **`facebook`**: This tensor was created from real data taken from the article "On the Evolution of User Interaction in Facebook," by Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P.Gummadi. In Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09), August 2009.
4. **`test_1`**: This is a synthetic tensor for testing `perfCpAlsTT` and `perfCpOptTT`.
5. **`test_2`**: This is a synthetic tensor for testing `perfCpAlsTT` and `perfCpOptTT`.

5.1.3 Ensign-Py3/test/test_data

- **`csv2tensor`**: This data is used for testing `csv2tensor.py`.
    1. **`abc.csv`**: A very small toy dataset
    2. **`cab_data_small.csv`**: A reduced list of entries from `taxi.csv`
    3. **`conn.log`**: This real (anonymized) data is generated within Reservoir Labs using Reservoir's internal network traffic messages.
    4. **`conn_mini.log`**: A reduced list of entries from `conn.log`
    5. **`taxi.csv`**: Taxicab trip records provided by NYC

- **`sync_labels`**: This data is used for testing `sync_labels.py`.
    1. **`cyber`**: This real (anonymized) data is generated within Reservoir Labs using Reservoir's internal network traffic messages.

5.4 For Streaming Users

The streaming (`perfCpAprStreaming`) method calculates updates to existing CP decompositions. The motivation for this operation is that an update of an existing decomposition usually takes significantly less time than a new decomposition of the extended dataset. For

example, consider a day's worth of network traffic data that has been formed into a tensor and then decomposed with either a CP. If another hour's worth of network traffic comes in the next day, it would be very expensive to re-form the full tensor with this additional data and then perform the requisite decomposition over again. Streaming algorithms can approximate that new decomposition using only the old decomposition and the new data.

This approximate update is accurate when the new data is low rank and/or, as in the case of CP streaming decompositions, partially expressible using the components of the old decomposition. As an example, if the (old and new) network traffic data contains mostly one type of traffic, such as DNS requests, then the update can be captured using (non-temporal) components of the old data CP. If the data update contains new types of network traffic, for example a new system monitoring protocol, new SSH connections, etc. that were not captured in the first decomposition, then the streaming algorithms are capable of identifying, computing and adding those components to the decomposition as long as there are not too many of them (the update is low-rank).

The streaming method performs updates in an *incremental* fashion. For this purpose, the `<update_tensor_file>` must be a fully formed sparse tensor file, but should only include the extended elements. The update tensor file should have a nonzero number of additional entries.

5.4.1 Streaming CP Decompositions

The streaming method is useful when given additional data along an extension of one of the modes (e.g., a time mode in time series data). When using this method, the user specifies the approximate rank of the data update using the `<rank_update>` argument. This is used to calculate a CP decomposition of the update tensor. For example, assume a three-dimensional N_1 x N_2 x N_3 tensor has been decomposed with a CP decomposition of rank r_1. If the user specifies a rank-r_2 update, then the algorithm will first perform a rank-r_2 CP decomposition of the update data, and will use this information to update the old decomposition. This will result in an updated CP decomposition of rank (r_1+r_2) that will then be truncated back to rank r_1 using the updated weights for ordering.

To execute the CP streaming decomposition method (`perfCpAprStreaming`), the indexing of the streaming mode in the update tensor should start from 0, while the indexing of the non-streaming modes must match the original tensor. That is, the same index must be used for the same element (e.g., a specific IP address) across the two tensors. For tensors with associated mode maps, the `sync_stream.py` tool exists to provide this reconciliation. The `<rank_update>` argument must be a positive integer. Note that the streaming CP decomposition algorithm only allows for extension (streaming) along a single mode at a time.

For example, an original tensor file might read:

```
sptensor
3
105 60 15
1008
87    13    4     1
87    55    14    0
12    34    4     1
...
```

Whereas an update tensor with streaming mode 1 should read:

```
sptensor
3
105 20 15
418
99    0     7     5
10    17    14    9
1     5     0     1
...
```

In particular, note that the sizes along modes 0 and 2 match those of the original tensor. Also, the indexing of the second mode ranges from 0 to 19.

For tensors with associated mode maps, the `sync_stream.py` tool exists to simplify the reconciliation of the base and update tensors. Synchronization refers to reindexing the tensors and potential associated decompositions so that the index-label mapping is common to both the base decomposition and the update tensor along chosen (non-streaming) modes. This meets the requirement for performing a streaming update. See Section 4.2.5 for more details on the usage of `sync_stream.py`.

As an example, consider two tensors with associated mode maps stored in **`conn1`** and **`conn2`**, each encoding one of two consecutive days of network traffic. The modes represent timestamp, sender IP, receiver IP, and receiver port. Ultimately, we wish to stream along mode 0, that representing time. The first tensor would be decomposed by running:

```
$> perfCpApr conn1/tensor_data.txt 100 1 -o conn1
```

Now the first, second, and third modes of the second (update) tensor do not necessarily align with those of the first (base) tensor, as it is very likely that different sets of IP addresses were on the network each day. So to ensure that the update tensor file satisfies the requirements described above, run `sync_stream.py`:

```
$> sync_stream.py -m 0 -o sync conn1 conn2
```

This prepares the base decomposition (`conn1`) to have the update tensor (`conn2`) streamed onto it along mode 0 (specified by the `-m` flag). All of the modes other than the streaming mode (first, second, and third) are synchronized to have the same label maps. For the streaming mode (mode 0), entries with overlapping labels are removed from the update tensor. In this case, overlapping labels could occur if the update tensor overlaps the base tensor in time. These entries, converted to their labels for ease of reference, are captured in a file (`sync/update/dropped.csv`). The synchronized version of `conn1` and `conn2` are stored in `sync/base` and `sync/update` (specified by `-o` flag).

After proper synchronization, it is possible to run the streaming tool:

```
$> perfCpAprStreaming sync/base sync/update/tensor_data.txt 5 \
1 -o streamed
```

Finally, it is necessary to concatenate the mode maps of the zero-th modes and copy the other non-streaming mode maps. This is done by the `joinModeMaps.sh` script, which takes the base and update directories, as well as the streaming output directory, as arguments:

```
$> joinModeMaps.sh sync/base sync/update streamed
```

After this operation, the `streamed` directory contains a streaming decomposition with matching, merged mode maps. This decomposition can then be used as the base tensor decomposition and associated mode maps for further streaming updates.

6 Glossary

CANDECOMP/PARAFAC (CP) Decomposition: CANonical DECOMPosition (CANDECOMP), also known as PARAllel FACtor (PARAFAC) analysis, or CANDECOMP-PARAFAC (CP) for short. A rank-R CP decomposition is a way of rewriting a tensor as a sum of R rank-1 tensors weighted by an R-vector lambda.

Component: A single rank-1 tensor in the sum produced by a CP decomposition. A component is composed of one vector of scores for each mode and a weight (lambda).

Coordinate Format: A standard sparse tensor data structure available for use in all decompositions. The simple multi-dimensional extension of the sparse matrix coordinate format.

Cosine Similarity: Taken together with norm scaling, cosine similarity is an alternate (perhaps more intuitive) measure of fit for CP decompositions. Viewing the original tensor and the resulting approximate summed component decomposition as vectors, the cosine similarity is the cosine of the angle formed by the two vectors (a value between 1 and -1, with 1 indicating perfect alignment).

CP-ALS: The Alternating Least Squares method for computing a CP decomposition. This proceeds by iteratively fixing all but one factor matrix and then selecting the non-fixed factor matrix in this iteration to minimize a least squares norm.

CP-APR: The Alternating Poisson Regression method for computing a CP decomposition. By assuming random variation is selected from a Poisson distribution, CP-APR produces components with non-negative scores only.

CP-APR-PDNR: The Alternating Poisson Regression method for computing a CP decomposition using the Projected Damped Newton Row subproblem (PDN-R) algorithm. CP-APR-PDNR uses second derivatives in an attempt to improve convergence time (trading off longer per-iteration time for fewer total iterations) while also improving the expected sparsity of the solution.

CP-APR-PQNR: The Alternating Poisson Regression method for computing a CP decomposition using the Projected Quasi-Newton Row subproblem (PQN-R) algorithm. CP-APR-PQNR uses a limited-memory and faster approximation to second derivatives in an attempt to improve convergence time and memory usage (trading off greater total iterations for faster per-iteration time) while also improving the expected sparsity of the solution.

Dimension: see *Mode*

Element: One of the members of the domain of a single mode.

Entry: One of the values contained by a tensor, selected by a tuple of indices.

Factor Matrix: A matrix composed of the vectors from each component in a single mode. There will be one factor matrix for each mode of the original tensor. For example, for a rank-3 decomposition T = sum (a_i * b_i * c_i), there are three factor matrices: A = [a_1, a_2, …, a_R], B = [b_1, b_2, …, b_R], C = [c_1, c_2, …, c_R] where a_i, b_i, and c_i are the column vectors whose outer product is component i of the decomposition.

Fiber: Analogue of columns/rows of a matrix. Defined by fixing all indices except one, which is allowed to vary over all possible values. An n-mode fiber is composed of all entries sharing every index besides the index on the nth mode and forms a line of entries in the tensor. For example, T(i, :, k) is the 2-mode fiber with indices i and k on the first and third modes.

Fit: The CP decomposition is almost always an approximation. The fit measures how closely the CP decomposition approximates the original tensor. The resulting value will be a floating point value no greater than 1 (and can be less than zero in extreme cases), where closer to 1 means that the decomposition results form a tensor closer to the original data. For original tensor X and reconstructed CP decomposition P, the final fit is given as:

$$Fit = 1 - \frac{\sqrt{norm(X)^2 - 2(X \bullet P) + norm(P)^2}}{norm(X)}$$

Flattening: see *Matricization*.

Index: Accessors for each individual dimension of the tensor, analogous to the notation used in matrices. For example, i, j, k where T(i, j, k) is a single entry in a third-order tensor T.

KKT Violation: The extent to which the Karush-Kuhn-Tucker (KKT) conditions - necessary first-order conditions that the CP-APR optimization problem must satisfy - are violated. KKT Violation is the convergence condition for CP-APR, so lower is better and the iterative method ends when the KKT Violation is smaller than the specified stop tolerance.

Loglikelihood: The objective function of the CP-APR decomposition. CP ALS minimizes least-squared error (fit), assuming that the random variation in the tensor data follows a Gaussian distribution. For sparse count data, a Poisson distribution better captures the random variation, so we use loglikelihood instead. As presented in the output, larger values of loglikelihood are better.

Matricization: Otherwise known as flattening. This is a way of rewriting the entries of a higher-order tensor into a matrix, unfolding along certain modes. Typically this results in a matrix whose columns are the n-mode fibers laid out in some chosen order.

Mode: An individual dimension of a tensor. For example, consider a third-order tensor, *Location x Age x Date* where the entries are populations. Then the modes of this tensor are *Location*, *Age*, and *Date*, and *T*(New York,30,5/13/2000) would be the number of 30 year olds in New York on May 13, 2000.

Mode-Generic Format: A tensor data structure used only in the Tucker decomposition.
A generalization of the Coordinate format that handles semi-sparse tensors, that is a tensor with both dense and sparse modes.

Mode-Specific Format: A tensor data structure available for use in all decompositions. The mode-specific data structure contains a different copy of the tensor data for each dimension to provide better performance on tensors with diverse sparsity patterns.

Norm Scaling: Taken together with cosine similarity, norm scaling is an alternate (perhaps more intuitive) measure of fit for CP decompositions. Viewing the original tensor and the resulting approximate summed component decomposition as vectors, the norm scaling is the magnitude of the decomposition vector scaled relative to the original tensor vector (a non-negative value with 1 indicating equivalence).

Order: The number of dimensions (modes) that a tensor has. For example, a fifth-order tensor has five (5) dimensions, and each entry is specified by five (5) indices.

Rank-1 Tensor: A tensor that can be written as the generalized outer product of N vectors, where N is the order of the tensor. That is, if T is an Nth-order rank-1 tensor, there are N vectors v_1, ..., v_N such that

$$T(i_1, ..., i_N) = v_1(i_1) * ... * v_N(i_N)$$

for every tuple of indices i_1, ..., i_N. Each component of a CP decomposition is a rank-1 tensor. Note this definition generalizes from matrices where a rank-1 matrix can be thought of as the outer product of a row and column vector - each row of the resulting matrix is a scalar multiple (column vector entry) of the single row vector.

Score: Term used to refer to the value associated with a particular index in a particular mode of a CP decomposition component or a particular entry in a factor matrix. The score measures the degree to which an index of a particular mode is correlated with other high or low score indices in each mode within one component. The term *eigenscore* appears in some parts of the literature as an adaptation of the terms *eigenvalue* and *eigenvector* associated with matrices, although a score itself is not a direct multi-dimensional extension of either of these terms.

Slice: If a fiber is 1-d cross-section then a slice is a 2-d cross-section. Defined by fixing all indices except two, which are allowed to vary over all possible values, and results in a matrix.

Sparsity (Input): The input sparsity of the tensor decomposition is the number of zeros in the original tensor divided by the product of the sizes of the modes. ENSIGN includes optimizations to handle sparse inputs, reflecting the needs of real-world applications. Input tensors that are *extremely* sparse - e.g., 10^{-30} - can produce low quality results.

Sparsity (Output): The output sparsity is the sparsity of the resulting factor matrices: the number of zeros in the factor matrices divided by the sum of the sizes of the factor matrices. The first-order CP-APR tensor decomposition tends to underestimate the true sparsity of the factor matrices, so usually the second-order CP-APR-PDNR and CP-APR-PQNR have a higher sparsity. This corresponds to removing erroneous noise.

Streaming Decomposition (for select CP decompositions): A decomposition method that assumes the tensor consists of a base tensor, which has already been decomposed, summed with an update tensor, which is of low rank. It is possible to approximate the decomposition of the entire tensor by considering the base decomposition and the update without re-forming the full tensor and decomposing it.

Superdiagonal: The vector of entries where all the indices are the same. For example, for a third-order tensor T with modes of equal size I, then the superdiagonal is the vector <T(1,1,1), T(2,2,2) ,... , T(I,I,I)>. A tensor is superdiagonal if all of its nonzero entries are contained in its superdiagonal.

Tensor: A multidimensional array. A vector is a 1-d tensor, a matrix is a 2-d tensor; anything higher dimensional is typically called a "tensor"; for more precision, an N-d tensor can be referred to as an "Nth-order tensor".

Tensor n-mode matrix product: Multiplying a tensor by a matrix along a particular mode, n. Each n-mode fiber of the tensor is replaced by the result of multiplying it by the matrix.

Tensor Rank: The smallest number of rank-1 tensors whose sum is the tensor. It is possible to write a tensor of rank R as a sum with no more than R rank-1 tensors. To generate a compact approximation, the rank of a CP decomposition is often chosen to be less than the actual rank of the tensor.

Weights: Each component of a CP decomposition is assigned a weight, lambda. The weight represents how much of the variance of the original tensor this particular component explains. This is a coarse approximation of how important the component is.

7 Frequently Asked Questions

7.1 Understanding CP Decompositions

Q:
Which CP algorithm should I use to decompose a tensor?

A:
The best choice of algorithm generally depends on whether the entries of the tensor are derived from continuous (choose CP-ALS) or count (choose CP-APR, CP-APR-PDNR, or CP-APR-PQNR) values. As an example of continuous entries, consider the following tensor: a *location* x *humidity* x *light* tensor where the entries of the tensor are continuous temperature values. In this case, you should use CP-ALS. As an example of count entries, consider the following tensor: a *sender* x *receiver* x *date* tensor where the entries are a count of how many messages were sent and received on a particular date. In this case, you should use CP-APR,CP-APR-PDNR, or CP-APR-PQNR.

CP ALGORITHM	DATA TYPE BEST FOR	CAN DO JOINT DECOMP?	USE SECOND ORDER INFO?	NEGATIVE SCORES?
CP-ALS	Continuous	No	No	Yes
CP-ALS-NN	Continuous	No	No	No
CP-APR	Count	No	No	No
CP-APR-PDNR	Count	No	Yes	No
CP-APR-PQNR	Count	No	Yes	No

Note that missing entries are treated as zero when using CP-ALS for the decomposition of sparse continuous-value tensors derived from measurements. Future versions of ENSIGN may include alternate implementations that address missing entries rather than treating them as zero.

Q:
What is going on inside the CP decomposition algorithm?

A:
At its core, the CP decomposition algorithm is a model-fitting algorithm driven by gradient descent. The choice of rank defines a fixed set of parameters (the scores and weights). From an initial guess, the algorithm iterates until either the solution can no longer be improved or a fixed iteration limit is reached. Each iteration attempts to improve the fitness of the existing solution. Among the CP decomposition algorithms, alternating algorithms update the score parameters for one mode at a time rather than all at once. CP-APR-PDNR uses the second derivative to improve the rate of convergence - trading off a longer per-iteration time with a goal of requiring fewer total iterations. CP-ALS/CP-ALS-NN and CP-APR/CP-APR-PDNR/CP-APR-PQNR use different assumptions about the distribution of the data (Gaussian vs. Poisson) to guide the search and define the fitness metric. This is why CP-ALS and CP-ALS-NN tend to be better choices for continuous data and CP-APR, CP-APR-PDNR, and CP-APR-PQNR tend to be a better choices for count data.

Q:
Is the result of a CP Decomposition unique?

A:
Generally, no. The rank decomposition (the CP decomposition of rank R where R is the tensor rank of the original tensor) is guaranteed to be unique under simple rotations. All CP decompositions use gradient descent to improve the final solution from an initial guess. However, in practice, the choice of rank is not exactly the rank of the tensor. Additionally, because the CP algorithms are gradient descent methods, they suffer inherently from the possibility of becoming stuck in a local minimum; in such cases the results may vary significantly. (This would be indicated by differing final fit values, which suggest the use of a different rank.) Therefore, if the CP decomposition does not precisely recapture the original tensor, it will not be unique; it will depend on the choice of initial guess. Despite this, results often exhibit only small variations (e.g., the ordering of weights of components may vary and the precise members of components may change slightly).

Q:
What is the rank of my tensor? What rank CP Decomposition should I perform?

A:
In general it is not possible to analytically determine the rank of a tensor with a reasonable amount of computation (the problem of determining the rank of a tensor is NP-hard[7]). As a rule of thumb, the more information that is contained in a tensor, the more components the decomposition needs to approximate it well. It is often necessary to try a few guesses and evaluate how much the fit improves each time. If the fit only improves by a marginal amount, or it doesn't improve at all, then you have found an acceptable rank.

As a simple heuristic intended only to provide a good starting point, the default rank provided by the ENSIGN GUI is equal to the *number of modes x ln(number of entries)*.

Q:
In a component of a CP-ALS decomposition, a mode has one or more negative scores. What does this mean?

A:
The CP-ALS algorithm allows negative scores. This extra degree of freedom (vs. CP-APR, CP-APR-PDNR, and CP-APR-PQNR) sometimes allows a better final fit with the same or fewer number of components. While there is no specific semantic interpretation of negative scores, in most cases where there were no negative tensor entries, negative scores can be thought of as cancelling out part of one or more other components - a sort of refinement. When looking at negative scores, keep in mind that an even number of negative scores across modes in a component results in a positive total contribution to the tensor. For this reason, it is sometimes useful to select scores based on absolute value.

[7] J. Hastad, “Tensor rank is NP–complete,” J. of Algorithms, vol. 11, no. 4, pp. 644–654, 1990.

Q:
Why not use multidimensional k-means clustering to find patterns in data? Or Singular Value Decomposition (SVD) on a matrix?

A:
Multidimensional k-means clustering finds spatial clusters in multidimensional data. The clusters are limited to sets of points that are in close proximity according to some (usually Euclidean) distance metric. A meaningful distance metric may not readily exist for categorical data (e.g., names). Further, components from a CP decomposition frequently include multiple, spatially separated, high-score indices from one or more modes. If you interpret a CP component as a set of (partial or entire) data points constructed from the outer-product of these high-score indices, then there is no limit to the spatial distance between points in that set. Components that represent sets with widely separated data points (e.g., multiple IP senders sending to multiple receivers and ports) reveal patterns that cannot be recovered with k-means clustering.

The matrix SVD is a fairly powerful technique for extracting structure from data. The primary application of the SVD is to construct a new basis for a high-dimensional space, ordered by significance, and then to use that basis to perform a linear transformation of the high-dimensional data to a lower dimensional space. Matrix SVD methods then typically use some form of clustering or vector comparison (e.g., cosine similarity) in the lower dimensional space to identify similar data points. While the SVD, through the construction of the basis vectors, can identify latent features of interest, the transformation will still map similar points in the original dataset to similar points in the image. Thus, the method still tends to cluster based on proximity - although now in a way that is skewed for more significant features. While this skewing can cluster more widely separated data points, it will also tend to cluster any intermediate point on the line between them in the original high-dimensional space. In contrast, CP tensor decompositions can identify patterns made up of associations of discrete elements in the multimodal space.

A point of confusion sometimes arises from the fact that tensor decompositions originated as a generalization of the matrix SVD. One view of an SVD is as the decomposition of a matrix into a weighted ordered sum of rank-1 matrices (each the outer product of two vectors drawn from the decomposition matrices, scaled by a corresponding eigenvalue). The CP decomposition generalizes this concept - a tensor is decomposed into the the weighed sum of rank-1 components (each the outer product of two or more vectors). A Tucker decomposition further generalizes the matrix SVD by allowing nonzero off-diagonal entries in the sigma matrix (core tensor). However, in practical application, the CP decomposition components are not generally used as a basis for the tensor and there is not a simple transformation to map a new tensor entry to a linear combination of the components. Thus, a CP decomposition is not typically used the same way as an SVD is used and the two - while related - tend to have different strengths in practical application.

7.2 Building Tensors

Q:
I have a mode that is indexed by date/IP/some other form of category, instead of numerically, but the tensor file format requires the indices to be numbers only. How should I include this mode?

A:
The tensor format requires that the indices of each mode be integers. When a tensor is made, it is necessary to generate a mapping from categories (types) to indices. For example, if one mode of your tensor expresses IP, you will assign 0 to 10.0.0.1, 1 to 10.0.1.1, 2 to 10.1.1.255, etc. The indices can be sorted to maintain an original ordering if applicable. For example, for dates, it may be helpful to assign the indices so that the indices are in the same order as the dates-- for example 0 -> 12/2/2013, 1 -> 12/3/2013, 2 -> 12/4/2013, etc. The decomposition results will also be expressed using these indices. To interpret the meaning of an index, one should have saved the mapping - then the value of the index can be looked up in the map to obtain the original meaning.

The csv2tensor tool (see Sections 4.2.1 and 4.3.2) will also create these mappings from categories to indices and produce tensors that follow the mapping. The `type` flag can be used to specify the category (type) of each mode and therefore allow sorting for that category.

Q:
What are common mistakes that result in a tensor that is difficult to decompose with CP?

A:
The two most common mistakes are (1) constructing a tensor that is too sparse - that is, does not have enough entries relative to the product of the mode sizes and (2) constructing a tensor that is diagonal (or nearly diagonal) in one or more modes - that is, all (or most) of the elements in the mode(s) occur exactly once. Both cases result in a tensor that is likely to have a rank that is equal, or close to equal, to the number of entries. In these cases, there is a strong tendency toward the best-fit decomposition being one in which each component simply represents a single entry in the tensor. In the very sparse case, this happens because the entries are unlikely to share fibers with other entries. In the diagonal case, any component that has a high score for an element used only once can only have other high scores corresponding to the other elements included in that one entry. High scores anywhere else in the component contribute to error when the generalized outer product is considered - that is, add entries to the sum that do not exist in the original tensor.

Q:
What about very dense tensors?

A:
There is nothing wrong with using a dense tensor input, but notice that the runtime of the decompositions scale with the number of nonzeros. If the tensor is both large and fully dense, the number of nonzeros will be large. Tensors with hundreds of millions of nonzeros may take several hours to decompose.

Q:
How will ENSIGN behave if my dataset is incomplete or has missing or corrupted entries?

A:
Missing entries are handled automatically by the sparse tensor format. The sparse format is a list of nonzero entries, so it is not necessary to list every possible entry. However, entries not in the list are assumed to be zero.

For example, if we had the following dataset:

New York, Summer, High Elevation = 70 degrees
New York, Summer, Low Elevation = 90 degrees
New York, Winter, High Elevation = 10 degrees

Notice there's missing data here. We don't have the temperature for New York, Winter, and Low Elevation. With the sparse data format, the algorithm will treat <New York, Winter, Low> as zero degrees. For some datasets (e.g., network traffic datasets) this is a good thing. For a temperature datasets like this one, it can skew the results, so it's important to be careful during tensor construction.

Other types of incomplete/corrupted data can cause errors. For example, "NaN" or "Inf" within a dataset cannot be converted into an integer or float and thus will not be recognized by ENSIGN. During tensor building these need to be replaced with other values, binned separately, or removed.

Q:
Is it permissible to include two nonzeros with the exact same indices in a sptensor file (i.e., duplicate entries)? What about indices that don't appear in any entry in a sptensor file?

A:
The TTB function that converts sptensor files into data structures technically combines duplicate entries by summing their values. However, we have noticed that both CP-APR-PDNR and CP-APR-PQNR produce different results with duplicate entries compared to when these duplicate entries are pre-combined by summing the values, and so, for now, we recommend avoiding including duplicates. Likewise, all indices of each mode in an sptensor file should appear in at least one entry.

7.3 Dealing with Errors

Q:
My tensor decomposition is seg-faulting early in progress. What does this mean?

A:
We have observed that on some systems the terminal stack memory limits result in a segmentation fault. It is easily fixed by first running the command:

```
$>  ulimit -s unlimited
```

No other segmentation faults have been reported. This does not mean that the tensor is too big. If a tensor is too big to complete in a reasonable amount of time, the current implementation will simply continue to run. We have implemented a guard that warns when the input size may be too big to be reasonably kept in memory. The size of this guard can be controlled in the configuration file.

Q:
Under what conditions do CP-APR-PDNR or CP-APR-PQNR give a “Fatal Error”? What do these errors mean?

A:
CP-APR-PDNR attempts to invert a Hessian (matrix of mixed second partial derivatives) as part of trying to find a more effective direction to minimize the objective function. The algorithm is written so that if this matrix is singular (and thus non-invertible), the step calculation reverts to the first-order method. This condition generates an error in the matrix inversion routine, but is not, by itself, an error in CP-APR-PDNR.

In practice, most of the time when the Hessian is ill-conditioned, reverting to the first-order approach for a single iteration tends to increase the likelihood of a well-conditioned Hessian for the next several iterations. After passing these sporadic ill-conditioned iterations the algorithm will continue normally and will still produce good results (as indicated by the final fit).

CP-APR-PQNR faces a similar situation, as it also attempts to approximate the inverse of the Hessian, using the alternate method of L-BFGS, a popular optimization algorithm. When the true Hessian is ill-conditioned, the approximation will be ill-conditioned as well, and CP-APR-PQNR will exhibit the same behavior of sporadic ill-conditioned iterations.

An even more general problem, however, is that some tensors do not match the Poisson distribution for the random variance model that both second order CP-APR algorithms assume. A common example of this is a continuous-valued tensor. In these cases, there is no reason to expect that the Hessian matrix will be invertible. Reverting to the first-order method does not overcome this fundamental mismatch and the algorithms never converge. In these cases, the Hessian will continue to be singular for many consecutive iterations. After twenty (20) consecutive iterations with singular Hessians, the algorithms halt with a warning. The result is saved, but the final fit in these cases will most likely be very poor.

Ultimately, the exact conditions for when the Hessian will be non-invertible are extremely complicated: they depend on the original data, how well that data fits the APR model, the rank of the decomposition, and the initial guess. So, for any given input, it is not immediately possible to predict whether, or not, or how many, singular Hessians will occur. For this reason, it may be the case that the number of errors varies unpredictably with the number of components selected for the decomposition. This is not a cause for concern.

7.4 Other Questions

Q:
Is there a good starting-point reference for understanding the mathematics behind tensor decompositions?

A:
T. G. Kolda and B. W. Bader, Tensor Decompositions and Applications, SIAM Review 51(3):455-500, September 2009.

http://www.sandia.gov/~tgkolda/pubs/pubfiles/TensorReview.pdf