TensorClus

Release 0.0.2

Rafika Boutalbi

Feb 20, 2022

CONTENTS

TensorClus (Tensor Clustering) is the first Python library aiming to cluster and co-clustering tensor data. It allows to easily perform tensor clustering through decomposition or tensor learning and tensor algebra. TensorClus allows easy interaction with other python packages such as NumPy, Tensorly, TensorFlow, or TensorD, and run methods at scale on CPU or GPU.

It supports major operating systems namely Microsoft Windows, MacOS, and Ubuntu.

TensorClus is distributed under the 3-Clause BSD license. It works with Python>= 3.6.

Note: If you use this software as part of your research, please cite: *R. Boutalbi, L. Labiod, and M. Nadif. Tensorclus: A python library for tensor (co)-clustering.Neuro-computing, 468:464–468, 2022*

CHAPTER

ONE

INSTALLATION

You can install **TensorClus** with all the dependencies with:

```
pip install TensorClus
```

It will install the following libraries:

- numpy
- pandas
- scipy
- scikit-learn
- matplotlib
- coclust
- tensorly
- tensorflow

1.1 Install from GitHub repository

To clone TensorClus project from github:

```
# Install git LFS via https://www.atlassian.com/git/tutorials/git-lfs
# initialize Git LFS
git lfs install Git LFS initialized.
git init Initialized
# clone the repository
git clone https://github.com/boutalbi/TensorClus.git
cd TensorClus
# Install in editable mode with `-e` or, equivalently, `--editable`
pip install -e .
```

Note: The latest TensorClus development sources are available on https://github.com/boutalbi/TensorClus

1.2 Running the tests

In order to run the tests, you have to install nose, for example with:

```
pip install nose
```

You also have to get the datasets used for the tests:

```
git clone https://github.com/boutalbi/TensorClus.git
```

And then, run the tests:

```
cd cclust_package
nosetests --with-coverage --cover-inclusive --cover-package=TensorClus
```

CHAPTER

TWO

EXAMPLES

The datasets used here are available at:

https://github.com/boutalbi/TensorClus/tree/master/TensorClus/reader

2.1 Basic usage

In the following example, the DBLP1 dataset is loaded from the reader module. A tensor co-clustering is applied using the 'SparseTensorCoclusteringPoisson' algorithm with 3 clusters. The The accuracy measure is printed and the predicted row labels and column labels are retrieved for further exploration or evaluation.

```
import TensorClus.coclustering.sparseTensorCoclustering as tcSCoP
from TensorClus.reader import load
import numpy as np
from coclust.evaluation.external import accuracy

######################################################################
# Load DBLP1 dataset #
######################################################################
data_v2, labels, slices = load.load_dataset("DBLP1_dataset")
n = data_v2.shape[0]
######################################################################
# Execute TSPLBM on the dataset #
######################################################################

# Define the number of clusters K
K = 3
# Optional: initialization of rows and columns partitions
z=np.zeros((n,K))
z_a=np.random.randint(K,size=n)
z=np.zeros((n,K))+ 1.e-9
z[np.arange(n) , z_a]=1
w=np.asarray(z)

# Run TSPLBM

model = tcSCoP.SparseTensorCoclusteringPoisson(n_clusters=K , fuzzy = True,init_row=z,␣
↪init_col=w,max_iter=50)
model.fit(data_v2)
predicted_row_labels = model.row_labels_
```

(continues on next page)

(continued from previous page)

```
predicted_column_labels = model.column_labels_

acc = np.around(accuracy(labels, predicted_row_labels),3)
print("Accuracy : ", acc)
```

CHAPTER

THREE

TENSORCLUS READER

The `TensorClus.reader` module provides functions to load and read different data format.

`TensorClus.reader.load.`**`load_dataset`**`(`*datasetName*`)`
Load one of the available dataset.

datasetName [str] the name of dataset

tensor three-way numpy array

labels true row classes (ground-truth)

slices slices name

`TensorClus.reader.load.`**`read_txt_tensor`**`(`*filePath*`)`
read tensor data from text file.

filePath [str] the path of file

tensor three-way numpy array

`TensorClus.reader.load.`**`save_txt_tensor`**`(`*tensor*, *filePath*`)`
save tensor data as a text file.

tensor : tensor array

filePath [str] the path of file

CHAPTER

FOUR

TENSORCLUS DECOMPOSITION

The *TensorClus.decomposition.decomposition_with_clustering* module provides a class with common methods for multiple clustering alorihtm from decomposition results.

class TensorClus.decomposition.decomposition_with_clustering.**DecompositionWithClustering**(*n_clusters=[2, 2, 2]*, *modes=[1, 2, 3]*, *algorithm='Kmeans++*

Clustering from decomposition results.

n_clusters [array-like, optional, default: [2,2,2]] Number of row clusters to form

modes [array-like, optional, default: [1,2,3]] Selected modes for clustering

algorithm [string, optional, default: "kmeans++"] Selected algorithm for clustering

labels_ [array-like, shape (n_rows,)] clustering label of each row

fit(*X*, *y=None*)

Perform Tensor co-clustering.

X : decomposition results

CHAPTER

FIVE

TENSORCLUS COCLUSTERING

The *TensorClus.coclustering.sparseTensorCoclustering* module provides an implementation of a Sparse tensor co-clustering algorithm.

class TensorClus.coclustering.sparseTensorCoclustering.**SparseTensorCoclusteringPoisson**(*n_clusters=2*, *fuzzy=True*, *init_row=None*, *init_col=None*, *max_iter=50*, *n_init=1*, *tol=1e-06*, *random_state=None*, *gpu=None*)

Tensor Latent Block Model for Poisson distribution.

n_row_clusters [int, optional, default: 2] Number of row clusters to form

n_col_clusters [int, optional, default: 2] Number of column clusters to form

fuzzy [boolean, optional, default: True] Provide fuzzy clustering, If fuzzy is False a hard clustering is performed

init_row [numpy array or scipy sparse matrix, shape (n_rows, K), optional, default: None] Initial row labels

init_col [numpy array or scipy sparse matrix, shape (n_cols, L), optional, default: None] Initial column labels

max_iter [int, optional, default: 20] Maximum number of iterations

n_init [int, optional, default: 1] Number of time the algorithm will be run with different initializations. The final results will be the best output of *n_init* consecutive runs.

random_state [integer or numpy.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

tol [float, default: 1e-9] Relative tolerance with regards to criterion to declare convergence

row_labels_ [array-like, shape (n_rows,)] Bicluster label of each row

column_labels_ [array-like, shape (n_cols,)] Bicluster label of each column

gamma_kl [array-like, shape (k,l,v)] Value $\frac{p_{kl}}{p_{k.} \times p_{.l}}$ for each row cluster k and column cluster l

gamma_kl_evolution [array-like, shape(k,l,max_iter)] Value of gamma_kl of each bicluster according to iterations

F_c(*x*, *z*, *w*, *gammakl*, *pi_k*, *rho_l*, *choice='ZW'*)

Compute fuzzy log-likelihood (LL) criterion.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

gammakl [three-way numpy array, shape=(K,L, v_features)] matrix of bloc's parameters

pi_k [numpy array, shape(K,)] vector of row cluster proportion

rho_l [numpy array, shape(K,)] vector of column cluster proportion

choice [string, take values in ("Z", "W", "ZW")] considering the optimization of LL

(H_z, H_w, LL, value) (row entropy, column entropy, Log-likelihood, lower bound of log-likelihood)

fit(*X*, *y=None*)
Perform Tensor co-clustering.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

gammakl(*x*, *z*, *w*)
Perform Tensor co-clustering.

x [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z : row partition w : column partition Returns ——- gamma_kl_mat

three-way numpy array, shape=(K,L, v_features) Computed parameters per block

pi_k(*z*)
Compute row proportion.

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

pi_k_vect numpy array, shape=(K) proportion of row clusters

rho_l(*w*)
Compute column proportion.

w [numpy array, shape(d_col_objects, L)] matrix of column partition

rho_l_vect numpy array, shape=(L) proportion of column clusters

The *TensorClus.coclustering.tensorCoclusteringPoisson* module provides an implementation of a tensor co-clustering algorithm for count three-way tensor.

class TensorClus.coclustering.tensorCoclusteringPoisson.**TensorCoclusteringPoisson**(*n_row_clusters=2*, *n_col_clusters=2*, *fuzzy=True*, *init_row=None*, *init_col=None*, *max_iter=50*, *n_init=1*, *tol=1e-06*, *random_state=None*, *gpu=None*)

Tensor Latent Block Model for Poisson distribution.

n_row_clusters [int, optional, default: 2] Number of row clusters to form

n_col_clusters [int, optional, default: 2] Number of column clusters to form

fuzzy [boolean, optional, default: True] Provide fuzzy clustering, If fuzzy is False a hard clustering is performed

init_row [numpy array or scipy sparse matrix, shape (n_rows, K), optional, default: None] Initial row labels

init_col [numpy array or scipy sparse matrix, shape (n_cols, L), optional, default: None] Initial column labels

max_iter [int, optional, default: 20] Maximum number of iterations

n_init [int, optional, default: 1] Number of time the algorithm will be run with different initializations. The final results will be the best output of *n_init* consecutive runs.

random_state [integer or numpy.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

tol [float, default: 1e-9] Relative tolerance with regards to criterion to declare convergence

row_labels_ [array-like, shape (n_rows,)] Bicluster label of each row

column_labels_ [array-like, shape (n_cols,)] Bicluster label of each column

gamma_kl [array-like, shape (k,l,v)] Value $\frac{p_{kl}}{p_{k.} \times p_{.l}}$ for each row cluster k and column cluster l

gamma_kl_evolution [array-like, shape(k,l,max_iter)] Value of gamma_kl of each bicluster according to iterations

F_c(*x*, *z*, *w*, *gammakl*, *pi_k*, *rho_l*, *choice='ZW'*)

Compute fuzzy log-likelihood (LL) criterion.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

gammakl [three-way numpy array, shape=(K,L, v_features)] matrix of bloc's parameters

pi_k [numpy array, shape(K,)] vector of row cluster proportion

rho_l [numpy array, shape(K,)] vector of column cluster proportion

choice [string, take values in ("Z", "W", "ZW")] considering the optimization of LL

(H_z, H_w, LL, value) (row entropy, column entropy, Log-likelihood, lower bound of log-likelihood)

fit(*X*, *y=None*)

Perform Tensor co-clustering.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

gammakl(*x*, *z*, *w*)

Compute gamma_kl per bloc.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

gamma_kl_mat three-way numpy array, shape=(K,L, v_features) Computed parameters per block

pi_k(*z*)

Compute row proportion.

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

pi_k_vect numpy array, shape=(K) proportion of row clusters

rho_l(*w*)
Compute column proportion. Parameters ———- w : numpy array, shape(d_col_objects, L)

matrix of column partition

rho_l_vect numpy array, shape=(L) proportion of column clusters

The *TensorClus.coclustering.tensorCoclusteringGaussian* module provides an implementation of a tensor co-clustering algorithm for continous three-way tensor.

class TensorClus.coclustering.tensorCoclusteringGaussian.**TensorCoclusteringGaussian**(*n_row_clusters=2*, *n_col_clusters=2*, *fuzzy=True*, *parsimonious=True*, *init_row=None*, *init_col=None*, *max_iter=50*, *n_init=1*, *tol=1e-06*, *random_state=None*, *gpu=None*)

Tensor Latent Block Model for Normal distribution.

n_row_clusters [int, optional, default: 2] Number of row clusters to form

n_col_clusters [int, optional, default: 2] Number of column clusters to form

fuzzy [boolean, optional, default: True] Provide fuzzy clustering, If fuzzy is False a hard clustering is performed

parsimonious [boolean, optional, default: True] Provide parsimonious model, If parsimonious False sigma is computed at each iteration

init_row [numpy array or scipy sparse matrix, shape (n_rows, K), optional, default: None] Initial row labels

init_col [numpy array or scipy sparse matrix, shape (n_cols, L), optional, default: None] Initial column labels

max_iter [int, optional, default: 20] Maximum number of iterations

n_init [int, optional, default: 1] Number of time the algorithm will be run with different initializations. The final results will be the best output of *n_init* consecutive runs.

random_state [integer or numpy.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

tol [float, default: 1e-9] Relative tolerance with regards to criterion to declare convergence

row_labels_ [array-like, shape (n_rows,)] Bicluster label of each row

column_labels_ [array-like, shape (n_cols,)] Bicluster label of each column

mu_kl [array-like, shape (k,l,v)] Value :math: mean vector for each row cluster k and column cluster l

sigma_kl_ [array-like, shape (k,l,v,v)] Value of covariance matrix for each row cluster k and column cluster

F_c(*x*, *z*, *w*, *mukl*, *sigma_x_kl*, *pi_k*, *rho_l*, *choice='ZW'*)
Compute fuzzy log-likelihood (LL) criterion.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

mukl [three-way numpy array, shape=(K,L, v_features)] matrix of mean parameter pe bloc

sigma_x_kl [Four-way numpy array, shape=(K,L,v_features, v_features)] tensor of sigma matrices for all blocks

pi_k [numpy array, shape(K,)] vector of row cluster proportion

rho_l [numpy array, shape(K,)] vector of column cluster proportion

choice [string, take values in ("Z", "W", "ZW")] considering the optimization of LL

(H_z, H_w, LL, value) (row entropy, column entropy, Log-likelihood, lower bound of log-likelihood)

fit(*X*, *y=None*)
Perform Tensor co-clustering.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

mukl(*x*, *z*, *w*)
Compute the mean vector mu_kl per bloc.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

mukl_mat three-way numpy array, shape=(K,L, v_features) Computed parameters per block

pi_k(*z*)
Compute row proportion.

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

pi_k_vect numpy array, shape=(K) proportion of row clusters

rho_l(*w*)
Compute column proportion.

w [numpy array, shape(d_col_objects, L)] matrix of column partition

rho_l_vect numpy array, shape=(L) proportion of column clusters

sigma_x_kl(*x*, *z*, *w*, *mukl*)
Compute the mean vector sigma_kl per bloc.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

mukl [numpy array, shape(K,L, v_features)] tensor of mukl values

sigma_x_kl_mat three-way numpy array Computed the covariance parameters per block

The *TensorClus.coclustering.tensorCoclusteringBernoulli* module provides an implementation of a tensor co-clustering algorithm for binary three-way tensor.

class TensorClus.coclustering.tensorCoclusteringBernoulli.**TensorCoclusteringBernoulli**(*n_row_clusters=2*, *n_col_clusters=2*, *fuzzy=False*, *init_row=None*, *init_col=None*, *max_iter=50*, *n_init=1*, *tol=1e-06*, *random_state=None*, *gpu=None*)

Tensor Latent Block Model for Bernoulli distribution.

n_row_clusters [int, optional, default: 2] Number of row clusters to form

n_col_clusters [int, optional, default: 2] Number of column clusters to form

fuzzy [boolean, optional, default: True] Provide fuzzy clustering, If fuzzy is False a hard clustering is performed

init_row [numpy array or scipy sparse matrix, shape (n_rows, K), optional, default: None] Initial row labels

init_col [numpy array or scipy sparse matrix, shape (n_cols, L), optional, default: None] Initial column labels

max_iter [int, optional, default: 20] Maximum number of iterations

n_init [int, optional, default: 1] Number of time the algorithm will be run with different initializations. The final results will be the best output of *n_init* consecutive runs.

random_state [integer or numpy.RandomState, optional] The generator used to initialize the centers. If an integer is given, it fixes the seed. Defaults to the global numpy random number generator.

tol [float, default: 1e-9] Relative tolerance with regards to criterion to declare convergence

row_labels_ [array-like, shape (n_rows,)] Bicluster label of each row

column_labels_ [array-like, shape (n_cols,)] Bicluster label of each column

mu_kl [array-like, shape (k,l,v)] Value :math: mean vector for each row cluster k and column cluster l

F_c(*x*, *z*, *w*, *mukl*, *pi_k*, *rho_l*, *choice='ZW'*)

Compute fuzzy log-likelihood (LL) criterion.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

mukl [three-way numpy array, shape=(K,L, v_features)] matrix of mean parameter pe bloc

pi_k [numpy array, shape(K,)] vector of row cluster proportion

rho_l [numpy array, shape(K,)] vector of column cluster proportion

choice [string, take values in ("Z", "W", "ZW")] considering the optimization of LL

(H_z, H_w, LL, value) (row entropy, column entropy, Log-likelihood, lower bound of log-likelihood)

fit(*X*, *y=None*)
Perform Tensor co-clustering.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

mukl(*x*, *z*, *w*)
Compute the mean vector mu_kl per bloc.

X [three-way numpy array, shape=(n_row_objects,d_col_objects, v_features)] Tensor to be analyzed

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

w [numpy array, shape(d_col_objects, L)] matrix of column partition

mukl_mat three-way numpy array

pi_k(*z*)
Compute row proportion.

z [numpy array, shape= (n_row_objects, K)] matrix of row partition

pi_k_vect numpy array, shape=(K) proportion of row clusters

rho_l(*w*)
Compute column proportion.

w [numpy array, shape(d_col_objects, L)] matrix of column partition

rho_l_vect numpy array, shape=(L) proportion of column clusters

CHAPTER

SIX

TENSORCLUS VIZUALISATION

The `TensorClus.vizualisation` module provides functions to visualize different measures or data.

`TensorClus.vizualisation.__init__.`**`Plot_CoClust_axes_etiquette`**(*title*, *fig*, *axes*, *data*, *phiR*, *phiC*, *K*, *L*, *etiquette*)

Plot CoClustering results for each slice on specific axes.

title: title of figure

fig : figure that includes all axes

axes : list of axes corresponding to the number of slices

data : tensor data

phiR : row clustering partition

phiC : row clustering partition

K : number of row cluster

L : number of columns cluster

etiquette : name of slices

`TensorClus.vizualisation.__init__.`**`duplicates`**(*lst*, *item*)

Find index of duplicated values.

lst: list of values item: values to determine

list index of dipulicated values

`TensorClus.vizualisation.__init__.`**`generateColour`**()

Generate random color.

str hex color

`TensorClus.vizualisation.__init__.`**`plot_logLikelihood_evolution`**(*model*, *do_plot=True*, *save=False*, *dpi=200*)

Plot all intermediate loglikelihood for a model at each iteration.

model: `TensorClus.coclustering`, Fitted model

do_plot: boolean, Whether the plot should be displayed. True by default. Disabling this allows users to handle displaying the plot themselves.

save : boolean, False by default. Allowing save plot as image

dpi : int, 200 by default. Allowing to choose a specific resolution when saving image

TensorClus.vizualisation.__init__.**plot_parameter_evolution**(*model*, *do_plot=True*, *save=False*, *dpi=200*)

Plot all intermediate gammaKK parameters for a model at each iteration.

model: `TensorClus.coclustering`, Fitted model

do_plot: boolean, Whether the plot should be displayed. True by default. Disabling this allows users to handle displaying the plot themselves.

save : boolean, False by default. Allowing save plot as image

dpi : int, 200 by default. Allowing to choose a specific resolution when saving image

TensorClus.vizualisation.__init__.**plot_slice_reorganisation**(*data*, *model*, *slicesName=None*, *do_plot=True*, *save=False*, *dpi=200*)

Plot all intermediate modularities for a model.

data : tensor data

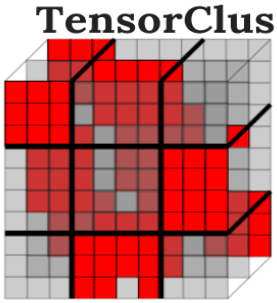
model: `TensorClus.coclustering.CoclustMod`, Fitted model

slicesName : list of slice names

do_plot: boolean, Whether the plot should be displayed. True by default. Disabling this allows users to handle displaying the plot themselves.

save : boolean, False by default. Allowing save plot as image

dpi : int, 200 by default. Allowing to choose a specific resolution when saving image



PYTHON MODULE INDEX

t

INDEX

S

T