libscientific Documentation

Release 1.2.4

Giuseppe Marco Randazzo

Jun 03, 2022

CONTENTS

CHAPTER

ONE

WHAT IS LIBSCIENTIFIC?

Libscientific aims to do numerical/multivariate and statistical analysis. The library is written in C language for almost all the computations except for a few methods such as the singular value decomposition (SVD) and the eigenvector/eigenvalue transformation from a third-party library, the Lapack library. It requires two dependencies: CMake and a c/fortran compiler and supports Windows, Linux, macOS, and embedded systems. Python bindings are available using ctypes to avoid dependencies. The library is distributed under GPLv3 license allowing it to be used for public and commercial purposes.

The significant advantage of libscientific is that it does not require dependencies unless a c/fortran compiler and CMake. Moreover, the size of the library is under 1.5MB. Libscientific also has been tested with old computers.

1.1 What can I do with libscientific?

- **Multivariate analysis:**
  - Principal Component Analysis using the NIPALS algorithm
  - Partial Least Squares (PLS) using the NIPALS algorithm (Regression and Classification)
  - Multiple Linear Regression (MLR) using the Ordinary Least Squares algorithm
  - Linear Discriminant Analysis (LDA) using the Fisher algorithm
  - Multi-waw unfolding PCA (UPCA) using the S. Wold, P. Geladi and K. Esbensen algorithm
  - Multi-way unfolding PLS (UPLS) using the S. Wold, P. Geladi and K. Esbensen algorithm
- **Matrix/Vector/Tensor computations:**
  - Matrix/Vector and Vector/Matrix dot product
  - Matrix/Matrix dot product
  - Vector/Vector dot product
  - Matrix inversion using the Gauss-Jordan algorithm or the LU decomposition
  - Matrix pseudo inversion using the Moore-Penrose formula
  - Matrix transpose
  - Tensor/Tensor dot product
  - Vector/Matrix Kronecker product
  - Tensor/Matrix dot product
  - Tensor transpose
- **Statistical analysis:**

  - Descriptive statistics of a matrix
  - R squared (R^2)
  - Mean absolute error (MAE)
  - Mean squared error (MSE)
  - Root mean squared error (RMSE)
  - Bias estimation (BIAS)
  - Sensitivity binary classification test
  - Positive predicted values binary classification test
  - Receiver operating characteristic curve (ROC)
  - Precision-Recal curve (PR)
- **Design of experiment:**
  - Bifactorial matrix expansion used in design of experiments (DOE)
  - Yates analysis to assess the variable effects in a DOE
- **Clustering:**
  - K-Means with different initialization (Random, ++ and so on) for divisive clustering
  - Hierarchical clustering for aglomerative clustering
  - Hyper grid map which create clusters dividing the hyperspace into a hyper grid and abundance score plot
- **Instance/Object selection:**
  - Most descriptive compound selection
  - Maximum dissimilarity selection
- **Optimization:**
  - Downhill optimization using the Nelder–Mead method aka Simplex method
- **Interpolation:**
  - Natural cubic spline interpolation
- **Similarity analysis:**
  - Euclidean distance analysis
  - Manhattan distance analysis
  - Cosine distance analysis
- **Model validation:**
  - Leave-One-Out
  - K-Fold cross validation
  - Repeated K-Fold cross validation
  - Train/Test split
- **Variable selection:**
  - Genetic agorithm variable selection for PLS only

- Particle Swarm Optimization variable selection for PLS only
- Spearman Ranking variable selection for PLS only

1.2 History

- 2009: Developed and used during my PhD thesis at the Laboratory of chemometrics and cheminformatics at the University of Perugia
- 2016: Open-source the code release under GPLv3. The development still continue under open-source licence

1.3 Integration with other open-source projects

- Libscientific is the engine of QStudioMetrics, a software to develop easy multivariate analysis
- MolecularFramework a c/c++ cheminformatic library to analyze 3D molecules and develop process 3D information for model development

1.4 Licensing

CHAPTER

TWO

INSTALL

The installation works for Linux/macOS and Windows.

2.1 Requirements

- A development environment (On windows msys/msys2 or visual studio)
- A c/fortran compiler
- Cmake
- python3 (if you whant to use the library in python)

2.2 Installation process

First you need to install the C library following these instructions:

```
git clone https://github.com/gmrandazzo/libscientific.git
cd libscientific
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/usr/local/ ..
make -j5
sudo make install
```

Then, if you want to use the library in python, you have also to install the python package.

```
cd ../src/python_bindings
python3 setup.py install

or

pip3 install libscientific
```

2.3 Packages

On macOS you can install libscientific via homebrew

```
brew install --HEAD libscientific
brew install --HEAD libscientific-python3
```

CHAPTER

THREE

GETTING STARTED IN C

Every data type object in libscientific is stored in the HEAP and then supports dynamic memory allocation.

Hence, every data object such as matrix, vectors, tensors, models, in general, need to be manually allocated/deallocated by the programmer using the library predefined constructs “NewSOMETHING(&…);” and “DelSOMETHING(&…);”.

Hence, every data object is a pointer and needs to pass by reference “&”. To avoid memory fragmentation problems, please, consider deallocating every allocated variable at the end of your program :-)

3.1 Compile a program that use libscientific

A program that use libscientific requires only one directive as follow:

```
#include <scientific.h>
```

and to compile the code with a C or C++ compiler linking with *-lscientific* and specify the right paths using the -*L<library path/of/libscientific* and -*I<include path/of libscientific*> options.

```
gcc -o example1 -L/usr/local/lib/ -I/usr/local/include -lscientific example1.c
```

CHAPTER

FOUR

VECTOR OPERATIONS

4.1 Create/Allocate a vector

There are four different types of vectors

- Double vector: dvector
- Integer vector: ivector
- Unsigned integer vector: uivector
- String vector: strvector

Here we show an example on how to allocate/deallocate these four vector types.

```c
#include <stdio.h>
#include <scientific.h>


int main(void){
  int i;
  /* Define the variable vector, which in this case is a double vector.
   * If we would like to use an integer or an unsigned integer or a
   * string vector instead this construct became
   * ivector *v; or uivector *v; or strvector *v;
   */

  dvector *v;
  NewDVector(&v, 5); // Allocate the memory space.

  // Fill the value inside the vector
  for(i = 0; i < 5; i++){
    v->data[i] = (float)i;
  }

  // Show the vector values to video
  PrintDVector(v);

  // Free the memory space
  DelDVector(&v);
}

```

4.2 Append a value to a given vector

Here we show an example on how to append a value to a vector.

```c
#include <stdio.h>
#include <scientific.h>


int main(void){
  int i;

  /*
   * We define here the double vector.
   * If we would like to utilize another vector
   * we can change dvector with the other three possibilities:
   * - uivector
   * - ivector
   * - strvector
   *
   */
  dvector *v;

  NewDVector(&v, 5); // We intialize the vector

  // We add 5 numbers
  for(i = 0; i < 5; i++){
    v->data[i] = (float)i;
  }

  // We append a new number
  DVectorAppend(&v, 123.4);

  // Print to video the result
  PrintDvector(v);

  // Free up the memory
  DelDVector(&v);
}

```

4.3 Work with string vectors

CHAPTER

FIVE

MATRIX OPERATIONS

Matrix is a user-defined data type that contains: - the number of rows - the number of columns - the 2D data array, which defines the matrix.

The data array is selected explicitly as a double type to work with an extensive range of numbers.

```
typedef struct{
  double **data;
  size_t row, col;
}matrix;
```

5.1 Create/Allocate a matrix with a specific size

Create a simple matrix of 10 rows and 15 columns and fill it with numbers.

Then print it to the terminal using "PrintMatrix();"

```
#include <stdio.h>
#include <scientific.h>


int main(void)
{
    int i, j;
    matrix *m; // Definition of the pointer matrix variable
    NewMatrix(&m, 10, 15); // Create the matrix with 10 rows and 15 columns. Each value in the matrix is 0.
    for(i = 0; i < 10; i++){
        for(j = 0; j < 15; j++){
            m->data[i][j] = (float)i+j; // Fill the matrix values with these numbers
        }
    }
    PrintMatrix(m); // Print to video the matrix content
    DelMatrix(&m); // Free the memory space
}
```

5.2 Initialize an empty matrix and append a row/column to it

An empty matrix is an object with rows and cols equal to 0. However in that matrix object we can add dynamically rows and columns and or resize it later on.

In this example we will initialize an empty matrix and we will add to it several rows.

```c
#include <stdio.h>
#include <scientific.h>

int main(void)
{
    int i;
    matrix *mx; // Definition of the matrix variable as a pointer
    dvector *row; // Definition of the row variable as a pointer

    NewDVector(&row, 15);
    for(i = 0; i < row->size; i++){
        row->data[i] = (double)i; // Fill one time the row vector
    }

    initMatrix(&mx); // Initialize the empty matrix with rows and columns equal to 0

    for(i = 0; i < 5; i++){
        MatrixAppendRow(&mx, row); // Append 5 times the row to the matrix mx
    }

    PrintMatrix(mx); // Print to video the matrix

    DelDVector(&row); // Free the memory space for the row vector
    DelMatrix(&mx); // Free the memory space for the matrix
}


```

Of course the same code can be reused to add a columns using the construct “MatrixAppendCol” instead of “MatrixAppendRow”.

5.3 Matrix x Column vector dot product

In this example we illustrate the product between a matrix of sizes M x N and a column double vector of size N.

```c
#include <stdio.h>
#include <scientific.h>

int main(void)
{
    int i, j;
    matrix *mx; // Definition of the matrix variable as a pointer
    dvector *cvect; // Definition of the column vector as a pointer
    dvector *result; // Definition of the result vector between the matrix and the␣
↪column vector
```

(continues on next page)

(continued from previous page)

```

    NewDVector(&cvect, 15);
    for(i = 0; i < cvect->size; i++){
        cvect->data[i] = (double)i; // Fill one time the row vector
    }

    NewMatrix(&mx, 23, 15); // Initialize a matrix with 23 rows and 15 columns

    for(i = 0; i < mx->row; i++){
        for(j = 0; j < mx->col; j++){
            mx->data[i][j] = (double)i+j;
        }
    }
    NewDVector(&result, 23);

    MatrixDVectorDotProduct(mx, cvect, result);
    /*
     * or MT_MatrixDVectorDotProduct if you want to run the multitask operation.
     * This function is usefull for large matrix.
     */


    PrintDVector(result); // Print to video the result
    // Free the memory spaces
    DelDVector(&result);
    DelDVector(&cvect);
    DelMatrix(&mx);
}


```

5.4 Transpose a matrix

A matrix transpose is an operation that flips a matrix over its diagonal. Here is an example that shows how to produce a transpose of a given matrix.

```
#include <stdio.h>
#include <scientific.h>


int main(void)
{
    int i, j;
    matrix *m, *m_T; // Definition of the pointer matrix variable
    NewMatrix(&m, 10, 15); // Create the matrix with 10 rows and 15 columns. Each value␣
↪in the matrix is 0.
    NewMatrix(&m_T, m->col, m->row); // Create the transposed matrix with the flip of␣
↪the columns and rows size

    for(i = 0; i < 10; i++){
```

(continues on next page)

(continued from previous page)

```
        for(j = 0; j < 15; j++){
            m->data[i][j] = (float)i+j; // Fill the matrix values with these numbers
        }
    }
    MatrixTranspose(m, m_T);
    PrintMatrix(m); // Print to video the original matrix content
    puts("Transposed matrix");
    PrintMatrix(m_T); // Print to video the transposed matrix
    // Free the memory spaces
    DelMatrix(&m);
    DelMatrix(&m_T);
}
```

5.5 Invert a matrix

In this example we show how to invert a matrix with libscientific

```
#include <stdio.h>
#include <scientific.h>
#include <math.h>

int main(void)
{
    int i;
    matrix *m; // Definition of the matrix variable as a pointer
    matrix *m_inv; // Definition of the inverted matrix variable as a pointer

    NewMatrix(&m, 10, 10); // Allocate the matrix to invert
    MatrixInitRandomFloat(m, -3., 3.); // Random fill the matrix with values within a␣
↪range -3 < x < 3
    PrintMatrix(m); // Print to video the matrix

    initMatrix(&m_inv); // Initialize the matrix to invert
    MatrixInversion(m, &m_inv); // Invert the matrix

    double det = fabs(MatrixDeterminant(m)); // Calculate the determinant

    printf("Determinant %.4f\n", det); // Print to video the matrix determinant
    PrintMatrix(m_inv); // Print to video the inverted matrix

    // Free the memory spaces
    DelMatrix(&m_inv);
    DelMatrix(&m);
}
```

5.6 Calculate eigenvectors and eigenvalues of a matrix

This example shows how to calculate eigenvectors and eigenvalues of an N x N real nonsymmetric matrix. The eigenvector/eigenvalue is computed thanks to the dgeev.f code extracted from the Lapack library.

```
#include <stdio.h>
#include <scientific.h>
#include <math.h>

int main(void)
{
    int i;
    matrix *m; // Definition of the matrix variable as a pointer
    dvector *eval; // Definition the variable to store the eigenvalues
    matrix *evect; // Definition of the variable were to store the eigenvectors

    NewMatrix(&m, 10, 10); // Allocate the matrix to invert
    MatrixInitRandomFloat(m, -3., 3.); // Random fill the matrix with values within a range -3 < x < 3
    PrintMatrix(m); // Print to video the matrix

    // Initialize the variables
    initDVector(&eval);
    initMatrix(&evect);

    EVectEval(m, &eval, &evect); // Calculate the eigenvectors and associated eigenvalues

    PrintDVector(eval); // Print to video the eigenvalues
    PrintMatrix(evect); // Print to video the eigenvectors. Each column correspond to an eingenvalue

    // Free the memory spaces
    DelDVector(&eval);
    DelMatrix(&evect);
    DelMatrix(&m);
}
```

CHAPTER

SIX

TENSOR OPERATIONS

TO BE COMPLETED

CHAPTER

SEVEN

MULTIVARIATE ANALYSIS ALGORITHMS

In this section, you will find examples of running multivariate analysis algorithms. In particular, the algorithm described here is extracted from official scientific publications and is adapted to run in multithreading to speed up the calculation.

- PCA and PLS implements the NIPALS algorithm described in the following publication:

P. Geladi, B.R. Kowalski
Partial least-squares regression: a tutorial
Analytica Chimica Acta Volume 185, 1986, Pages 1-17
DOI:10.1016/0003-2670(86)80028-9

7.1 Principal Component Analysis (PCA)

Here is an example that shows how to compute a principal component analysis on a matrix.

```
#include <stdio.h>
#include <scientific.h>

int main(void)
{
    matrix *m; // Definition of the input matrix
    PCAMODEL *model; // Definition of the PCA model
    int i, j;
    int nobj = 20;
    int nvars = 8;
    NewMatrix(&m, nobj, nvars);

    // Fill with random values the matrix m
    srand(nobj);
    for(size_t i = 0; i < nobj; i++){
        for(size_t j = 0; j < nvars; j++){
            m->data[i][j] = randDouble(0,20);
        }
    }


    NewPCAModel(&model); // Allocation of the PCA model
    PCA(m, 1, 5, model, NULL); // Calculation of the PCA on matrix m using unit variance
↪scaling (1) and the extraction of 5 principal components
```

(continues on next page)

(continued from previous page)

```

    PrintPCA(model); // Print to video the PCA results

    /* Of course you can print in a separate way the different results contained in the model variable
      * model->scores is the matrix of scores
      * model->loadings is the matrix of loadings
      * model->colavg is the column average obtained from the input matrix
      * model->scaling is the scaling factor obtained from the input matrix
      */

    // Free the memory spaces
    DelPCAModel(&model);
    DelMatrix(&m);
}

```

7.2 Partial Least Squares (PLS)

A matrix of features or independent vIariables and a matrix of targets or dependent variables is requested to calculate a PLS model. Here is a simple example that shows how to calculate a PLS model.

```
#include <stdio.h>
#include <scientific.h>

int main(void)
{
    matrix *x, *y; // Define the feature matrix x and the target to predict y
    dvector *betas; // Define the beta coefficients
    PLSMODEL *m;

    // Allocate the matrix
    NewMatrix(&x, 14, 6);
    NewMatrix(&y, 14, 1);

    // Fill the matrix with values
    // This is a manual filling.
    // Of course we can read a csv file and fill it automatically

    x->data[0][0] = 4.0000;  x->data[0][1] = 4.0000;  x->data[0][2] = 1.0000;  x->data[0][3] = 84.1400;  x->data[0][4] = 1.0500;  x->data[0][5] = 235.1500;
    x->data[1][0] = 5.0000;  x->data[1][1] = 5.0000;  x->data[1][2] = 1.0000;  x->data[1][3] = 79.1000;  x->data[1][4] = 0.9780;  x->data[1][5] = 231;
    x->data[2][0] = 4.0000;  x->data[2][1] = 5.0000;  x->data[2][2] = 1.0000;  x->data[2][3] = 67.0900;  x->data[2][4] = 0.9700;  x->data[2][5] = 249.0000;
    x->data[3][0] = 4.0000;  x->data[3][1] = 4.0000;  x->data[3][2] = 1.0000;  x->data[3][3] = 68.0700;  x->data[3][4] = 0.9360;  x->data[3][5] = 187.3500;
    x->data[4][0] = 3.0000;  x->data[4][1] = 4.0000;  x->data[4][2] = 2.0000;  x->data[4][3] = 68.0800;  x->data[4][4] = 1.0300;  x->data[4][5] = 363.0000;
    x->data[5][0] = 9.0000;  x->data[5][1] = 7.0000;  x->data[5][2] = 1.0000;  x->data[5][3] = 129.1600;  x->data[5][4] = 1.0900;  x->data[5][5] = 258.0000;
```

(continues on next page)

(continued from previous page)

```
    x->data[6][0] = 10.0000;  x->data[6][1] = 8.0000;  x->data[6][2] = 0.0000;  x->data[6][3] = 128.1600;  x->data[6][4] = 1.1500;  x->data[6][5] = 352.0000;
    x->data[7][0] = 6.0000;  x->data[7][1] = 6.0000;  x->data[7][2] = 0.0000;  x->data[7][3] = 78.1118;  x->data[7][4] = 0.8765;  x->data[7][5] = 278.6400;
    x->data[8][0] = 16.0000;  x->data[8][1] = 10.0000;  x->data[8][2] = 0.0000;  x->data[8][3] = 202.2550;  x->data[8][4] = 1.2710;  x->data[8][5] = 429.1500;
    x->data[9][0] = 6.0000;  x->data[9][1] = 12.0000;  x->data[9][2] = 0.0000;  x->data[9][3] = 84.1600;  x->data[9][4] = 0.7800;  x->data[9][5] = 279.0000;
    x->data[10][0] = 4.0000;  x->data[10][1] = 8.0000;  x->data[10][2] = 1.0000;  x->data[10][3] = 72.1100;  x->data[10][4] = 0.8900;  x->data[10][5] = 164.5000;
    x->data[11][0] = 4.0000;  x->data[11][1] = 9.0000;  x->data[11][2] = 1.0000;  x->data[11][3] = 71.1100;  x->data[11][4] = 0.8660;  x->data[11][5] = 210.0000;
    x->data[12][0] = 5.0000;  x->data[12][1] = 11.0000;  x->data[12][2] = 1.0000;  x->data[12][3] = 85.1500;  x->data[12][4] = 0.8620;  x->data[12][5] = 266.0000;
    x->data[13][0] = 5.0000;  x->data[13][1] = 10.0000;  x->data[13][2] = 1.0000;  x->data[13][3] = 86.1300;  x->data[13][4] = 0.8800;  x->data[13][5] = 228.0000;

    y->data[0][0] = 357.1500;
    y->data[1][0] = 388.0000;
    y->data[2][0] = 403.0000;
    y->data[3][0] = 304.5500;
    y->data[4][0] = 529.0000;
    y->data[5][0] = 510.0000;
    y->data[6][0] = 491.0000;
    y->data[7][0] = 353.3000;
    y->data[8][0] = 666.6500;
    y->data[9][0] = 354.0000;
    y->data[10][0] = 339.0000;
    y->data[11][0] = 360.0000;
    y->data[12][0] = 379.0000;
    y->data[13][0] = 361.0000;

    // Allocate the PLS model
    NewPLSModel(&m);

    /* Calculate the partial least squares algorithm taking as input:
     * x: the feature matrix x
     * y: the target matrix y
     * nlv: the number of latent variable nlv
     * xautoscaling: the autoscaling type for the x matrix
     * yautoscaling: the autoscaling type for the y Matrix
     * model: the PLSMODEL previously allocated
     * ssignal: a scientific signal to stop the calculation if requested by the user
     *
     * more information in the pls.h header file
     * void PLS(matrix *mx, matrix *my, size_t nlv, size_t xautoscaling, size_t yautoscaling, PLSMODEL *model, ssignal *s);
     */
    PLS(x, y, 3, 1, 0, m, NULL);

    PrintPLSModel(m); // Print to video the PLS model

```

(continues on next page)

(continued from previous page)

```
    /*Validate the model using the internal validation method*/
    MODELINPUT minpt; // Define the model input for the validation method
    minpt.mx = &x;
    minpt.my = &y;
    minpt.nlv = 3;
    minpt.xautoscaling = 1;
    minpt.yautoscaling = 0;

    // Use the boot strap random group cross validation.
    BootstrapRandomGroupsCV(&minpt, 3, 100, _PLS_, &m->predicted_y, &m->pred_residuals,␣
↪4, NULL, 0);
    // We can also compute leave one out in case...
    // LeaveOneOut(&minpt, _PLS_, &m->predicted_y, &m->pred_residuals, 4, NULL, 0);

    // Calculate the model validation statistics
    PLSRegressionStatistics(y, m->predicted_y, &m->q2y, &m->sdep, &m->bias);
    //Print to video the results of the validation and the predicted values

    puts("Q2 Cross Validation");
    PrintMatrix(m->q2y);
    puts("SDEP Cross Validation");
    PrintMatrix(m->sdep);
    puts("BIAS Cross Validation");
    PrintMatrix(m->bias);;

    // Calculate the beta coefficients to see the importance of each feature
    puts("Beta coefficients");
    initDVector(&betas);
    PLSBetasCoeff(m, GetLVCCutoff(m->q2y), &betas); // GetLVCCutoff select the best Q2␣
↪value from all the possibilities
    PrintDVector(betas);

    puts("PREDICTED VALUES");
    PrintMatrix(m->predicted_y);

    puts("PREDICTED RESIDUALS");
    PrintMatrix(m->pred_residuals);

    puts("REAL Y");
    PrintMatrix(y);

    // Free the memory spaces
    DelDVector(&betas);
    DelPLSModel(&m);
    DelMatrix(&x);
    DelMatrix(&y);
}
```

CHAPTER

EIGHT

GETTING STARTED IN PYTHON

Every data type object in libscientific is stored in the HEAP and then supports dynamic memory allocation.

In python, there is no need to allocate/deallocate matrix/vectors/tensors and models in general because the python binding itself automatically handles them.

8.1 Use libscientific in python

First, you need to install the c library and the python package. Please follow the process described here.

A program that use libscientific requires to import the python binding as follows

```
import libscientific
...
```

CHAPTER

NINE

VECTOR OPERATIONS

9.1 Create a vector in python

There are four different types of vectors

- Double vector: dvector
- Integer vector: ivector
- Unsigned integer vector: uivector
- String vector: strvector

Here we show an example on how create these four vector types.

```
#!/usr/bin/env python3
import libscientific
from random import random

# Create a list of values that you whant to convert to a double vector
a = [random() for j in range(5)]

# Transform the list a into a double vector d
d = libscientific.vector.DVector(a)

# Just print to video the content of vector d
d.debug()

# If you want to catch the value in position 1
print(d[1])

# If you want to modify the value in position 1
d[1] = -2

#If you want to get back the result as a "list"
dlst = d.tolist()

for item in dlst:
    print(item)
```

9.2 Append a value to a given vector

Here we show an example on how to append a value to a vector.

```
#!/usr/bin/env python3
import libscientific
from random import random

# Create a list of values that you whant to convert to a double vector
a = [random() for j in range(5)]
d = libscientific.vector.DVector(a)
# print the output of the double vector d
print("orig vector")
d.debug()


# append the value 0.98765 at the end of d
d.append(0.98765)
print("append 0.98765 at the end")
d.debug()

# extend the vector d with more other values from a list
d.extend([0.4362, 0.34529, 0.99862])
print("extent the vector with 3 more values")
d.debug()
```

CHAPTER

TEN

MATRIX OPERATIONS

Matrix is a user-defined data type that contains information in regards to - the number of rows - the number of columns - the 2D data array which defines the matrix

The data array in python uses the c language implementation. However, memory allocation/destruction is carried out directly from the python class. Hence there is no need to free up the memory manually.

10.1 Create a matrix in python

In this example, we show how to create a matrix from a list of lists (or numpy array), modify its content and convert it again to a list of lists.

```python
#!/usr/bin/env python3
import libscientific
from random import random

# Create a random list of list
a = [[random() for j in range(2)] for i in range(10)]

# Convert the list of list matrix into a libscientific matrix
m = libscientific.matrix.Matrix(a)

# Get the value at row 1, column 1
print("Get value example")
print(m[1, 1])

# Modify the value at row 1, column 1
print("Set value example")
m[1, 1] = -2.
m.debug()


# Convert the matrix again to a list of list
mlst = m.tolist()
for row in mlst:
    print(row)
```

CHAPTER ELEVEN

TENSOR OPERATIONS

TO BE COMPLETED

CHAPTER

TWELVE

MULTIVARIATE ANALYSIS ALGORITHMS

In this section, you will find examples of running multivariate analysis algorithms. In particular, the algorithm described here is extracted from official libscientific publications and is adapted to run in multithreading to speed up the calculation.

- PCA and PLS implements the NIPALS algorithm described in the following publication:

P. Geladi, B.R. Kowalski
Partial least-squares regression: a tutorial
Analytica Chimica Acta Volume 185, 1986, Pages 1–17
DOI:10.1016/0003-2670(86)80028-9

12.1 Principal Component Analysis (PCA)

Here is an example that shows to compute a principal component analysis on a matrix.

```
#!/usr/bin/env python3

import libscientific
import random

def mx_to_video(m, decimals=5):
    for row in m:
        print("\t".join([str(round(x, decimals)) for x in row]))

random.seed(123456)

# Create a random matrix of 10 objects and 4 features
a = [[random.random() for j in range(4)] for i in range(10)]
print("Original Matrix")
mx_to_video(a)

# Compute 2 Principal components using the UV scaling (unit variance scaling)
model = libscientific.pca.PCA(scaling=1, npc=2)
# Fit the model
model.fit(a)

# Show the scores
print("Showing the PCA scores")
```

(continues on next page)

(continued from previous page)

```
scores = model.get_scores()
mx_to_video(scores, 3)

# Show the loadings
print("Showing the PCA loadings")
loadings = model.get_loadings()
mx_to_video(loadings, 3)

# Show the explained variance
print(model.get_exp_variance())

# Reconstruct the original PCA matrix from the 2 principal components
print("Reconstruct the original PCA matrix using the PCA Model")
ra = model.reconstruct_original_matrix()
mx_to_video(ra)
```

12.2 Partial Least Squares (PLS)

A matrix of features or independent variables and a matrix of targets or dependent variables is requested to calculate a PLS model.

Here is a simple example that shows how to calculate a PLS model.

```
#!/usr/bin/env python3

import libscientific
import random

def mx_to_video(m, decimals=5):
    for row in m:
        print("\t".join([str(round(x, decimals)) for x in row]))

random.seed(123456)
x = [[random.random() for j in range(4)] for i in range(10)]
y = [[random.random() for j in range(1)] for i in range(10)]
xp = [[random.random() for j in range(4)] for i in range(10)]

print("Original Matrix")
print("X")
mx_to_video(x)
print("Y")
mx_to_video(y)
print("XP")
mx_to_video(xp)
print("Computing PLS ...")
model = libscientific.pls.PLS(nlv=2, xscaling=1, yscaling=0)
model.fit(x, y)
print("Showing the PLS T scores")
tscores = model.get_tscores()
mx_to_video(tscores, 3)
```

(continues on next page)

(continued from previous page)

```

print("Showing the PLS U scores")
uscores = model.get_uscores()
mx_to_video(uscores, 3)

print("Showing the PLS P loadings")
ploadings = model.get_ploadings()
mx_to_video(ploadings, 3)

print("Showing the X Variance")
print(model.get_exp_variance())


print("Predict XP")
py, pscores = model.predict(xp)
print("Predicted Y for all LVs")
mx_to_video(py, 3)
print("Predicted Scores")
mx_to_video(pscores, 3)
```

CHAPTER

THIRTEEN

LICENSING

Libscientific is distributed under GPLv3 license. To know more in details how the licens work please read the file "LICENSE" or go to "http://www.gnu.org/licenses/gpl-3.0.en.html"

CHAPTER FOURTEEN

INDICES AND TABLES

- genindex
- modindex
- search