SMURFF

Release v0.15.5-4-g802e74fa

Mar 19, 2021

Contents

Welcome to SMURFF's documentation. SMURFF is a highly optimized and parallelized framework for Bayesian Matrix and Tensors Factorization.

The easiest way to install SMURFF is to use conda:

```
conda install -c vanderaa smurff
```

To get started with SMURFF using IPython notebooks, have a look at our *Getting Started using IPython Notebooks*. To learn more about Matrix Factorization, have a look at *What is SMURFF*.

CHAPTER 1

Contents

1.1 What is SMURFF

SMURFF is a highly optimized and parallelized framework for Bayesian Matrix and Tensors Factorization.

1.1.1 What is Bayesian Matrix Factorization

Matrix factorization is a common machine learning technique for recommender systems, like books for Amazon or movies for Netflix.

The idea of these methods is to approximate the user-movie rating matrix R as a product of two low-rank matrices U and V such that R U × V . In this way U and V are constructed from the known ratings in R, which is usually very sparsely filled. The recommendations can be made from the approximation U × V which is dense. If M × N is the dimension of R then U and V will have dimensions M × K and N × K. Noise is added to the model to avoid overfitting.

1.1.2 What can SMURFF do for you

SMURFF supports multiple matrix factorization methods:

- BPMF, the basic version;
- Macau, adding support for high-dimensional side information to the factorization;
- GFA, doing Group Factor Anaysis.

Macau and BPMF can also perform **tensor** factorization.

1.2 Getting Started using IPython Notebooks

This section contains documentation generated from IPython notebooks that discuss different aspects of SMURFF.

Note: This page was generated from notebooks/a_first_example.ipynb.

1.2.1 A first example running SMURFF

In this notebook we will run the BPMF algorithm using SMURFF, on compound-activity data.

Downloading the data files

In these examples we use ChEMBL dataset for compound-proteins activities (IC50). The IC50 values and ECFP fingerprints can be downloaded using this smurff function:

[ ]:
```
import smurff

ic50_train, ic50_test, ecfp = smurff.load_chembl()
```

The resulting variables are all `scipy.sparse` matrices: `ic50` is a sparse matrix containing interactions between chemical compounds (in the rows) and protein targets (called essays - in the columns). The matrix is already split in as train and test set.

The `ecfp` contains compound features. These features will not be used in this example.

Having a look at the data

The `spy` function in `matplotlib` is a handy function to plot sparsity pattern of a matrix.

[ ]:
```
%matplotlib inline

from matplotlib.pyplot import figure, show

fig = figure()
ax = fig.add_subplot(111)
ax.spy(ic50_train.tocsr()[0:1000,:].T, markersize = 1)
show()
```

Running SMURFF

Finally we run make a BPMF training session and call `run`. The `run` function builds the model and returns the `predictions` of the test data.

[ ]:
```
session = smurff.BPMFSession(
                       Ytrain     = ic50_train,
                       Ytest      = ic50_test,
                       num_latent = 16,
                       burnin     = 40,
                       nsamples   = 200,
                       verbose    = 1,
                       checkpoint_freq = 1,
                       save_freq = 1,)

predictions = session.run()
```

We can use the `calc_rmse` function to calculate the RMSE.

```
[ ]: rmse = smurff.calc_rmse(predictions)
     rmse
```

Plotting predictions versus actual values

Next to RMSE, we can also plot the predicted versus the actual values, to see how well the model performs.

```
[ ]: %matplotlib notebook

     import numpy
     from matplotlib.pyplot import subplots, show

     y = numpy.array([ p.val for p in predictions ])
     predicted = numpy.array([ p.pred_avg for p in predictions ])

     fig, ax = subplots()
     ax.scatter(y, predicted, edgecolors=(0, 0, 0))
     ax.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=4)
     ax.set_xlabel('Measured')
     ax.set_ylabel('Predicted')
     show()
```

```
[ ]:
```

Note: This page was generated from notebooks/input_matrices_and_tensors.ipynb.

```
[ ]: import numpy as np
     import scipy.sparse as sp
     import smurff
```

1.2.2 Input to SMURFF

In this notebook we will look at how to provide input to SMURFF with dense and sparse matrices;

SMURFF accepts the following matrix files for train, test and side-info data:

- for dense matrix or tensor input: numpy.ndarrays
- for sparse matrices input: scipy Sparse matrices in COO, CSR or CSC format
- for sparse tensors: a wrapper around a pandas.DataFrame

Let's have a look on how this could work.

Dense Train Input

```
[ ]: # dense input
     Ydense  = np.random.rand(10, 20)
     session = smurff.TrainSession(burnin = 5, nsamples = 5)
     session.addTrainAndTest(Ydense)
     session.run()
```

Sparse Matrix Input

The so-called *zero* elements in sparse matrices can either represent

1. missing values, also called ‘unknown’ or ‘not-available’ (NA) values.
2. actual zero values, to optimize the space that stores the matrix

Important: * when calling `addTrainAndTest(Ytrain, Ytest, is_scarce)` the `is_scarce` refers to the `Ytrain` matrix. `Ytest` is *always* scarce. * when calling `addSideInfo(mode, sideinfoMatrix)` with a sparse `sideinfoMatrix`, this matrix is always fully known.

[ ]:
```
# sparse matrix input with 20% zeros (fully known)
Ysparse = sp.rand(15, 10, 0.2)
session = smurff.TrainSession(burnin = 5, nsamples = 5)
session.addTrainAndTest(Ysparse, is_scarce = False)
session.run()
```

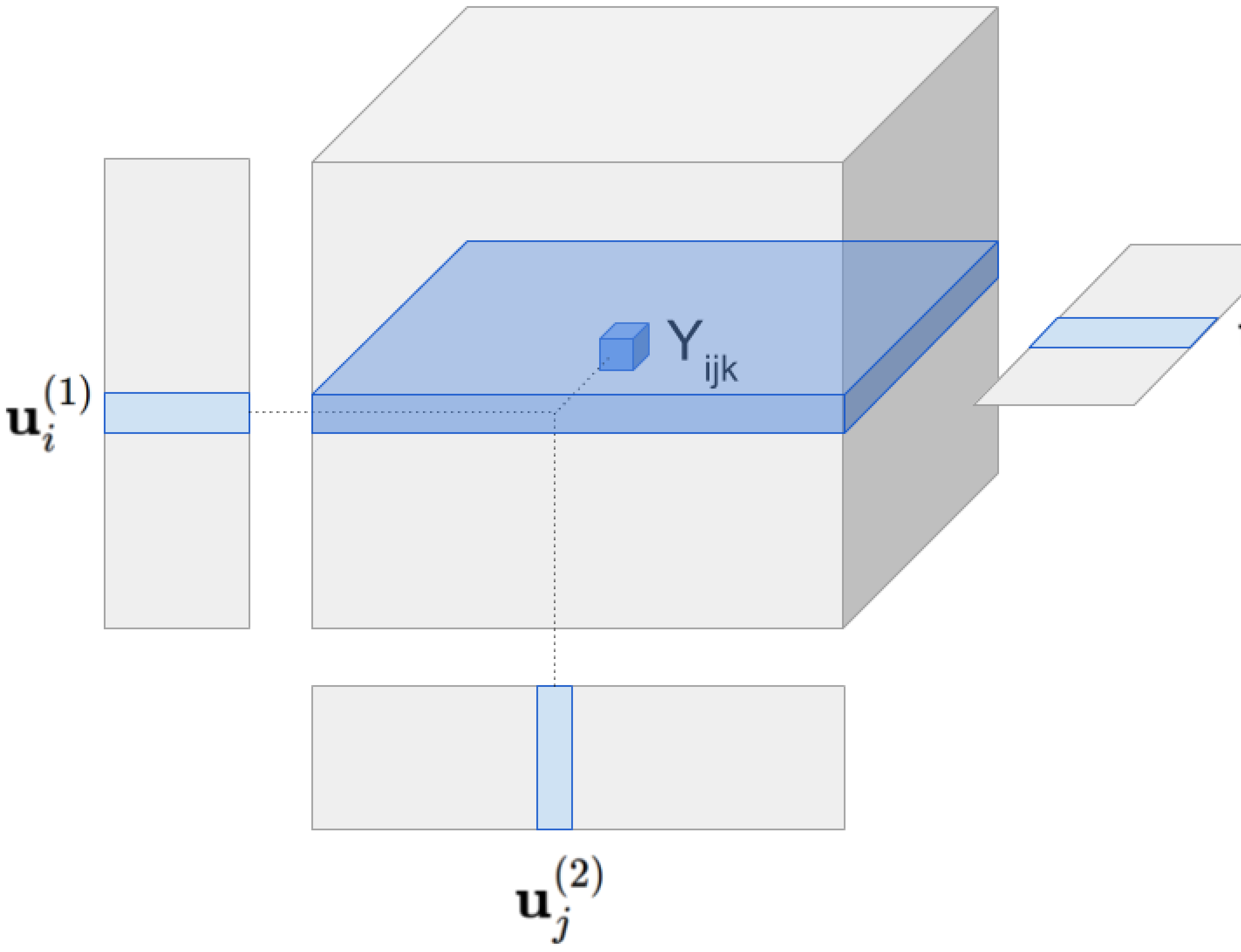
[ ]:
```
# sparse matrix input with unknowns (the default)
Yscarce = sp.rand(15, 10, 0.2)
session = smurff.TrainSession(burnin = 5, nsamples = 5)
session.addTrainAndTest(Yscarce, is_scarce = True)
session.run()
```

Tensor input

SMURFF also supports tensor factorization with and without side information on any of the modes. Tensor can be thought as generalization of matrix to relations with more than two items. For example 3-tensor of `drug x cell x gene` could express the effect of a drug on the given cell and gene. In this case the prediction for the element `Yhat[i,j,k]`* is given by

$$\hat{Y}_{ijk} = \sum_{d=1}^{D} u^{(1)}_{d,i} u^{(2)}_{d,j} u^{(3)}_{d,k} + mean$$

Visually the model can be represented as follows:



Tensor model predicts `Yhat[i,j,k]` by multiplying all latent vectors together element-wise and then taking the sum along the latent dimension (figure omits the global mean).

For tensors SMURFF implements a `SparseTensor` class. `SparseTensor` is a wrapper around a pandas `DataFrame` where each row stores the coordinate and the value of a known cell in the tensor. Specifically, the integer columns in the DataFrame give the coordinate of the cell and `float` (or double) column stores the value in the cell (the order of the columns does not matter). The coordinates are 0-based. The shape of the `SparseTensor` can be provided, otherwise it is inferred from the maximum index in each mode.

Here is a simple toy example with factorizing a 3-tensor with side information on the first mode.

[ ]:
```
import numpy as np
import pandas as pd
import scipy.sparse
import smurff
import itertools

## generating toy data
A = np.random.randn(15, 2)
B = np.random.randn(3, 2)
```

(continues on next page)

(continued from previous page)

```
C = np.random.randn(2, 2)

idx = list( itertools.product(np.arange(A.shape[0]),
                              np.arange(B.shape[0]),
                              np.arange(C.shape[0])) )
df  = pd.DataFrame( np.asarray(idx), columns=["A", "B", "C"])
df["value"] = np.array([ np.sum(A[i[0], :] * B[i[1], :] * C[i[2], :]) for i in idx ])

## assigning 20% of the cells to test set
Ytrain, Ytest = smurff.make_train_test_df(df, 0.2)

print("Ytrain = ", Ytrain)

## for artificial dataset using small values for burnin, nsamples and num_latents is␣
↪fine
predictions = smurff.BPMFSession(
                        Ytrain=Ytrain,
                        Ytest=Ytest,
                        num_latent=4,
                        burnin=20,
                        nsamples=20).run()

print("First prediction of Ytest tensor: ", predictions[0])
```

[ ]:

```

```

Note: This page was generated from notebooks/different_methods.ipynb.

1.2.3 Trying different Matrix Factorzation Methods

In this notebook we will try out several MF methods supported by SMURFF.

Downloading the files

As in the previous example we download the ChemBL dataset. The resulting IC50 matrix is a compound x protein matrix, split into train and test. The ECFP matrix has features as side information on the compounds.

[ ]:

```
import smurff

ic50_train, ic50_test, ecfp = smurff.load_chembl()
print(ic50)
```

Matrix Factorization without Side Information (BPMF)

As a first example we can run SMURFF without side information. The method used here is BPMF.

Input matrix for `Y` is a sparse scipy matrix (either coo_matrix, csr_matrix or csc_matrix). The test matrix `Ytest` also needs to ne sparse matrix of the same size as `Y`. Here we have used burn-in of 20 samples for the Gibbs sampler and then collected 80 samples from the model. We use 16 latent dimensions in the model.

For good results you will need to run more sampling and burnin iterations (>= 1000) and maybe more latent dimensions.

We create a session, and the `run` method returns the predictions of the `Ytest` matrix. `predictions` is a list of of type `Prediction`.

```
[ ]: session = smurff.BPMFSession(
                            Ytrain     = ic50_train,
                            Ytest      = ic50_test,
                            num_latent = 16,
                            burnin     = 20,
                            nsamples   = 80,
                            verbose    = 0,)

     predictions = session.run()
     print("First prediction element: ", predictions[0])

     rmse = smurff.calc_rmse(predictions)
     print("RMSE =", rmse)
```

Matrix Factorization with Side Information (Macau)

If we want to use the compound features we can use the Macau algorithm.

The parameter `side_info = [ecfp, None]` sets the side information for rows and columns, respectively. In this example we only use side information for the compounds (rows of the matrix).

```
[ ]: predictions = smurff.MacauSession(
                            Ytrain     = ic50_train,
                            Ytest      = ic50_test,
                            side_info  = [ecfp, None],
                            num_latent = 16,
                            burnin     = 40,
                            nsamples   = 100).run()

     smurff.calc_rmse(predictions)
```

Macau univariate sampler

SMURFF also includes an option to use a *very fast* univariate sampler, i.e., instead of sampling blocks of variables jointly it samples each individually. An example:

```
[ ]: predictions = smurff.MacauSession(
                            Ytrain     = ic50_train,
                            Ytest      = ic50_test,
                            side_info  = [ecfp, None],
                            univariate = True,
                            num_latent = 32,
                            burnin     = 500,
                            nsamples   = 3500,
                            verbose    = 1,).run()
     smurff.calc_rmse(predictions)
```

When using it we recommend using larger values for `burnin` and `nsamples`, because the univariate sampler mixes slower than the blocked sampler.

[ ]:
```

```

Note: This page was generated from notebooks/different_noise_models.ipynb.

1.2.4 Different noise models

In this notebook we look at the different noise models.

Prepare train, test and side-info

We first need to download and prepare the data files. This can be acomplished using this a built-in function is smurff. IC50 is a compound x protein matrix, The ECFP matrix as features as side information on the compounds.

[ ]:
```
import smurff

ic50_train, ic50_test, ecfp = smurff.load_chembl()
```

Fixed noise

The noise model of observed data can be annotated by calling `addTrainAndTest` with the optional parameter `noise_model`. The default for this parameter is `FixedNoise` with precision 5.0

[ ]:
```
session = smurff.TrainSession(
                            priors = ['normal', 'normal'],
                            num_latent=32,
                            burnin=100,
                            nsamples=500)

# the following line is equivalent to the default, not specifing noise_model
session.addTrainAndTest(ic50_train, ic50_test, smurff.FixedNoise(5.0))
predictions = session.run()
print("RMSE = %.2f" % smurff.calc_rmse(predictions))
```

Adaptive noise

Instead of a fixed precision, we can also allow the model to automatically determine the precision of the noise by using `AdaptiveNoise`, with signal-to-noise ratio parameters `sn_init` and `sn_max`.

- `sn_init` is an initial signal-to-noise ratio.
- `sn_max` is the maximum allowed signal-to-noise ratio. This means that if the updated precision would imply a higher signal-to-noise ratio than `sn_max`, then the precision value is set to `(sn_max + 1.0) / Yvar` where `Yvar` is the variance of the training dataset `Y`.

[ ]:
```
session = smurff.TrainSession(
                            priors = ['normal', 'normal'],
                            num_latent=32,
                            burnin=100,
                            nsamples=500)
```

(continues on next page)

(continued from previous page)

```
session.addTrainAndTest(ic50_train, ic50_test, smurff.AdaptiveNoise(1.0, 10.))
predictions = session.run()
print("RMSE = %.2f" % smurff.call_rmse(predictions))
```

Binary matrices

SMURFF can also factorize binary matrices (with or without side information). The input matrices can contain arbitrary values, and are converted to 0's and 1' by means of a threshold. To factorize them we employ probit noise model `ProbitNoise`, taking this `threshold` as a parameter.

To evaluate binary factorization, we recommed to use *ROC AUC*, which can be enabled by providing a threshold also to the `TrainSession`.

```
[ ]: ic50_threshold = 6.
     session = smurff.TrainSession(
                                 priors = ['normal', 'normal'],
                                 num_latent=32,
                                 burnin=100,
                                 nsamples=100,
                                 # Using threshold of 6. to calculate AUC on test data
                                 threshold=ic50_threshold)

     ## using activity threshold pIC50 > 6. to binarize train data
     session.addTrainAndTest(ic50_train, ic50_test, smurff.ProbitNoise(ic50_threshold))
     predictions = session.run()
     print("RMSE = %.2f" % smurff.calc_rmse(predictions))
     print("AUC = %.2f" % smurff.calc_auc(predictions, ic50_threshold))
```

The input train and test sets are converted to -1 and +1 values, if the original values are below or above the threshold (respectively). Similarly, the resulting predictions will be negative, if the model predicts the value to be below the threshold, or positive, if the model predicts the value to be above the threshold.

```
[ ]: predictions[1:10]
```

Binary matrices with Side Info

It is possible to enhance the model for binary matrices by adding side information using the Macau algorithm. Note that the *binary* here refers to the train and test data, not to the side information.

```
[ ]: ic50_threshold = 6.
     session = smurff.TrainSession(
                                 priors = ['macau', 'normal'],
                                 num_latent=32,
                                 burnin=100,
                                 nsamples=100,
                                 # Using threshold of 6. to calculate AUC on test data
                                 threshold=ic50_threshold)

     ## using activity threshold pIC50 > 6. to binarize train data
     session.addTrainAndTest(ic50_train, ic50_test, smurff.ProbitNoise(ic50_threshold))
     session.addSideInfo(0, ecfp, direct = True)
     predictions = session.run()
     print("RMSE = %.2f" % smurff.calc_rmse(predictions))
     print("AUC = %.2f" % smurff.calc_auc(predictions, ic50_threshold))
```

```
[ ]:
```

Note: This page was generated from notebooks/inference_with_smurff.ipynb.

1.2.5 Inference with SMURFF

In this notebook we will continue on the first example. After running a training session again in SMURFF, we will look deeper into how to use SMURFF for making predictions.

To make predictions we recall that the value of a tensor model is given by a tensor contraction of all latent matrices. Specifically, the prediction for the element $\hat{Y}_{ijk}$ of a rank-3 tensor is given by

$$\hat{Y}_{ijk} = \sum_{d=1}^{D} u^{(1)}_{d,i} u^{(2)}_{d,j} u^{(3)}_{d,k} + mean$$

Since a matrix is a rank-2 tensor the prediction for a matrix is given by:

$$\hat{Y}_{ij} = \sum_{d=1}^{D} u^{(1)}_{d,i} u^{(2)}_{d,j} + mean$$

These inner products are computed by SMURFF automagicaly, as we will see below.

Saving models

We run a `Macau` training session using side information (`ecfp`) from the chembl dataset. We make sure we *save every 10th sample*, such that we can load the model afterwards. This run will take some minutes to run.

```
[ ]: import smurff
import os

ic50_train, ic50_test, ecfp = smurff.load_chembl()

os.makedirs("ic50-macau", exist_ok=True)
session = smurff.MacauSession(
                       Ytrain     = ic50_train,
                       Ytest      = ic50_test,
                       side_info  = [ecfp, None],
                       num_latent = 16,
                       burnin     = 200,
                       nsamples   = 10,
                       save_freq  = 1,
                       save_prefix= "ic50-macau",
                       verbose    = 1,)

predictions = session.run()
```

Saved files

The saved files are indexed in a root ini-file, in this case the root ini-file will be `ic50-macau/root.ini`. The content of this file lists all saved info for this training run. For example

```
[options]
options = ic50-save-options.ini

[steps]
sample_step_10 = sample-10-step.ini
sample_step_20 = sample-20-step.ini
sample_step_30 = sample-30-step.ini
sample_step_40 = sample-40-step.ini
```

Each step ini-file contains the matrices saved in the step:

```
[models]
num_models = 2
model_0 = sample-50-U0-latents.ddm
model_1 = sample-50-U1-latents.ddm
[predictions]
pred = sample-50-predictions.csv
pred_state = sample-50-predictions-state.ini
[priors]
num_priors = 2
prior_0 = sample-50-F0-link.ddm
prior_1 = sample-50-F1-link.ddm
```

Making predictions from a `TrainSession`

The easiest way to make predictions is from an existing `TrainSession`:

```
[ ]: predictor = session.makePredictSession()
     print(predictor)
```

Once we have a `PredictSession`, there are serveral ways to make predictions:

- From a sparse matrix
- For all possible elements in the matrix (the complete $U \times V$)
- For a single point in the matrix
- Using only side-information

Predict all elements

We can make predictions for all rows $\times$ columns in our matrix

```
[ ]: p = predictor.predict_all()
     print(p.shape) # p is a numpy array of size: (num samples) x (num rows) x (num␣
     ↪columns)
```

Predict element in a sparse matrix

We can make predictions for a sparse matrix, for example our `ic50_test` matrix:

```
[ ]: p = predictor.predict_some(ic50_test)
     print(len(p),"predictions") # p is a list of Predictions
     print("predictions 1:", p[0])
```

Predict just one element

Or just one element. Let's predict the first element of our `ic50_test` matrix:

```
[ ]: from scipy.sparse import find
     (i,j,v) = find(ic50_test)
     p = predictor.predict_one((i[0],j[0]),v[0])
     print(p)
```

And plot the histogram of predictions for this element.

```
[ ]: %matplotlib inline
     import matplotlib.pyplot as plt

     # Plot a histogram of the samples.
     plt.subplot(111)
     plt.hist(p.pred_all, bins=10, density=True, label = "predictions's histogram")
     plt.plot(p.val, 1., 'ro', markersize =5, label = 'actual value')
     plt.legend()
     plt.title('Histogram of ' + str(len(p.pred_all)) + ' predictions')
     plt.show()
```

Make predictions using side information

We can make predictions for rows/columns not in our train matrix, using only side info:

```
[ ]: import numpy as np
     from scipy.sparse import find

     (i,j,v) = find(ic50_test)
     row_side_info = ecfp.tocsr().getrow(i[0])
     p = predictor.predict_one((row_side_info,j[0]),v[0])
     print(p)
```

Accessing the saved model itself

The latents matrices for all samples are stored in the `PredictSession` as `numpy` arrays

```
[ ]: # print the U matrices for all samples
     for i,s in enumerate(predictor.samples):
         print("sample", i, ":", [ (m, u.shape) for m,u in enumerate(s.latents) ])
```

This will allow us to compute predictions for arbitraty slices of the matrix or tensors using `numpy.einsum`:

```
[ ]: sample1 = predictor.samples[0]
     (U1, U2) = sample1.latents

     ## predict the slice Y[7, : ] from sample 1
     Yhat_7x = np.einsum(U1[:,7], [0], U2, [0, 2])

     ## predict the slice Y[:, 0:10] from sample 1
     Yhat_x10 = np.einsum(U1, [0, 1], U2[:,0:10], [0, 2])
```

The two examples above give a matrix (rank-2 tensor) as a result. It is adviced to make predictions on **all** samples, and average the predictions.

Making predictions from saved run

One can also make a `PredictSession` from a save root ini-file:

[ ]:
```
import smurff

predictor = smurff.PredictSession("ic50-macau/save-root.ini")
print(predictor)
```

[ ]:
```
```

Note: This page was generated from notebooks/centering.ipynb.

1.2.6 Centering

In this notebook we look at why it is important to center your matrices before using them with SMURFF.

SMURFF provides a function similar to sklearn.preprocessing.scale with the difference that SMURFF also supports scaling of sparse matrices. This makes sense only when the matrix is scarce, i.e. when the zero-elements represent unknown values.

The Matrix Factorization methods are meant for modeling the variance in the data. Hence it makes sense to subtract the mean first.

[ ]:
```
import smurff

ic50_train, ic50_test, ecfp = smurff.load_chembl()

ic50_train_centered, global_mean, _ = smurff.center_and_scale(ic50_train, "global",␣
↪with_mean = True, with_std = False)

ic50_test_centered = ic50_test
ic50_test_centered.data -= global_mean # only touch non-zeros
```

When we now run a SMURFF train session, we can see from the `Session` information, that the data has been centered:

```
PythonSession {
  Data: {
    Type: ScarceMatrixData [with NAs]
    Component-wise mean: 3.86555e-16
    ...
```

[ ]:
```
session = smurff.BPMFSession(
                       Ytrain     = ic50_train_centered,
                       Ytest      = ic50_test,
                       num_latent = 16,
                       burnin     = 40,
                       nsamples   = 200,
                       verbose    = 1,)

predictions = session.run()
rmse = smurff.calc_rmse(predictions)
print(rmse)
```

1.3 Installation

1.3.1 Installation using Conda

The easiest way to install SMURFF is to use Conda:

```
conda install -c vanderaa smurff
```

1.3.2 From source on Linux or MacOs

Building using cmake

Before continuing installation please check that

- cmake version is at least 3.6
- eigen3 version 3.3.3 or later is installed.

This is required due to the fixed Find scripts for BLAS libraries that are present in latest version.

Next, cmake has multiple switches:

- **Build type switches:**
  - CMAKE_BUILD_TYPE - Debug/Release
- **Algebra library switches (select only one):**
  - When no switches are specified, CMake will try to find any LAPACK and BLAS library on your system.
  - ENABLE_OPENBLAS - ON/OFF (should include openblas library when linking. openblas also contains implementation of lapack called relapack)
  - ENABLE_MKL - ON/OFF: tries to find the MKL single dynamic library.

Run CMake:

```
# install dependencies:
sudo apt-get install libopenblas-dev autoconf gfortran

# checkout and install Smurff
git clone https://github.com/ExaScience/smurff.git
cd smurff/lib/smurff-cpp/
mkdir build
cd build
cmake ../cmake -DENABLE_OPENBLAS=ON -DCMAKE_BUILD_TYPE=Debug
make
make test

# test the command-line program:
wget http://homes.esat.kuleuven.be/~jsimm/chembl-IC50-346targets.mm
Debug/smurff --train chembl-IC50-346targets.mm
```

The last command running smurff will be very slow, since we have compiled smurff in Debug mode.

Install command line smurff and Python package

Install:

```
make install
cd python/Smurff
python setup.py install
```

Test scripts for python are in *smurff/python/smurff/tests*. When the python package has been installed correctly, you can run *python -m unittest discover* in this directory.

1.3.3 From source on Windows

Before continuing installation please check that

- cmake version is at least 3.6
- Visual Studio version is 2013 or better 2015 because code uses some c++11 features

Install boost

Download latest version of boost from http://www.boost.org/

Start Visual Studio command prompt and execute the following commands.

```
bootstrap.bat
b2 toolset=msvc-14.0 address-model=64 --build-type=complete stage
```

This will compile and install boost libraries into build directory.

If you have Visual Studio different from 2015 - select proper toolset.

Configure boost environment variables as in the example:

```
BOOST_INCLUDEDIR=E:\boost_1_65_0
BOOST_LIBRARYDIR=E:\boost_1_65_0\stage\lib
```

Install eigen3

Execute the following commands from command prompt:

```
git clone https://github.com/RLovelett/eigen.git
cd eigen
mkdir build
cd build
cmake ../ -G "Visual Studio 14 2015 Win64"
```

If you have Visual Studio different from 2015 - select proper generator.

Build INSTALL target in Visual Studio in Release configuration.

This will build all projects and install them in Program Files by default.

Configure eigen3 environment variables as in the example:

```
EIGEN3_INCLUDE_DIR=C:\Program Files\Eigen3\include\eigen3
```

Install MinGW-64

MinGW-64 is required to build OpenBLAS library. MinGW-64 is chosen because it is easy to install fortran compiler dependency. Fortran compiler is requried for building ReLAPACK part of OpenBLAS. Other option (not described here) is to install fortran compiler directly. There are few binary distributions described here: http://fortranhelp.blogspot.ru/2010/09/i-have-just-installed-gfortran-on.html

Download installer at http://www.msys2.org/

Configure msys2 exactly as described in the guide.

Install corresponding packages with pacman

```
pacman -S gcc
pacman -S gcc-fortran
pacman -S make
pacman -S autoconf
pacman -S automake
```

Add path to MinGW-64 binaries to PATH variable as in the example:

```
C:\msys64\usr\bin
```

Install OpenBLAS

Open MinGW-64 command prompt

Execute the following commands:

```
git clone https://github.com/xianyi/OpenBLAS.git
cd OpenBLAS
make
make PREFIX=/e/openblas_install_64 install
```

You can change installation prefix if you want.

Set environment variables as in the example:

```
BLAS_INCLUDES=E:\openblas_install_64\include
BLAS_LIBRARIES=E:\openblas_install_64\lib\libopenblas.dll.a
```

Add path to OpenBLAS binaries as in the example:

```
E:\openblas_install_64\bin
```

Install Smurff

Execute the following commands from command prompt:

```
git clone https://github.com/ExaScience/smurff.git
cd smurff\lib\smurff-cpp\cmake
mkdir build
cd build
cmake ../ -G "Visual Studio 14 2015 Win64" -DENABLE_OPENBLAS=ON -DENABLE_VERBOSE_
↪COMPILER_LOG=ON
```

If you have Visual Studio different from 2015 - select proper generator.

Build INSTALL target in Visual Studio in Release configuration.

This will build all projects and install them in Program Files by default.

1.4 Python API Reference

This section of the documentation details modules, classed, and functions of SMURFF.

1.4.1 Preprocessing and Input Files

This section talks about:

Contents

Sparse Tensors

class smurff.**SparseTensor**(*data*, *shape=None*)

Wrapper around a pandas DataFrame to represent a sparse tensor

The DataFrame should have N index columns (int type) and 1 value column (float type) N is the dimensionality of the tensor

You can also specify the shape of the tensor. If you don't it is detected automatically.

Split Train and Test

smurff.**make_train_test**(*Y*, *ntest*)

Splits a sparse matrix Y into a train and a test matrix.

Parameters

- **Y** (*scipy sparse matrix (coo_matrix, csr_matrix or csc_matrix)*) – Matrix to split
- **ntest** (*float <1.0 or integer.*) –
  - if float, then indicates the ratio of test cells
  - if integer, then indicates the number of test cells

Returns

- **Ytrain** (*coo_matrix*) – train part
- **Ytest** (*coo_matrix*) – test part

Scaling and Centering

`smurff.center.center_and_scale(m, mode, with_mean=True, with_std=True)`

Center and/or scale the matrix m to the mean and/or standard deviation.

Parameters

- **m** (*{array-like, sparse matrix}*) – The data to center and scale.
- **mode** (*{ "rows", "cols", "global" }*) –
  - “rows”: center/scale each row indepently
  - ”cols”: center/scale each column idependently
  - ”global”: center/scale using global meand and/or standard deviation/
- **with_mean** (*boolean, True by default*) – If True, center the data before scaling.
- **with_std** (*boolean, True by default*) – If True, scale the data to unit variance (or equivalently, unit standard deviation).

Returns

- **m** (*array-like*) – Transformed array.
- **mean** (*array-like or double or None*) – Computed mean depending on mode
- **std** (*array-like or double or None*) – Computed standard deviation depending on mode

Notes

Also supports scaling of sparse matrices. This makes sense only when the matrix is scarce, i.e. when the zero-elements represent unknown values.

Example ChEMBL dataset

`smurff.load_chembl()`

Downloads a small subset of the ChEMBL dataset.

Returns

- **ic50_train** (*sparse matrix*) – sparse train matrix
- **ic50_test** (*sparse matrix*) – sparse test matrix
- **feat** (*sparse matrix*) – sparse row features

1.4.2 Training

Contents

– *StatusItem*

The most versatile class is `TrainSession`. `MacauSession` and `BPMFSession` provide a simpler interface.

TrainSession

class `smurff.`**`TrainSession`**(*priors=[u'normal', u'normal'], num_latent=NUM_LATENT_DEFAULT_VALUE, num_threads=NUM_THREADS_DEFAULT_VALUE, burnin=BURNIN_DEFAULT_VALUE, nsamples=NSAMPLES_DEFAULT_VALUE, seed=RANDOM_SEED_DEFAULT_VALUE, threshold=None, verbose=1, save_prefix=None, save_extension=None, save_freq=None, checkpoint_freq=None, csv_status=None*)

Class for doing a training run in smurff

A simple use case could be:

```
>>> session = smurff.TrainSession(burnin = 5, nsamples = 5)
>>> session.addTrainAndTest(Ydense)
>>> session.run()
```

`priors`
The type of prior to use for each dimension

Type list, where element is one of { "normal", "normalone", "macau", "macauone", "spikeandslab" }

`num_latent`
Number of latent dimensions in the model

Type int

`burnin`
Number of burnin samples to discard

Type int

`nsamples`
Number of samples to keep

Type int

`num_threads`
Number of OpenMP threads to use for model building

Type int

`verbose`
Verbosity level for C++ library

Type {0, 1, 2}

`seed`
Random seed to use for sampling

Type float

`save_prefix`
Path where to store the samples. The path includes the directory name, as well as the initial part of the file names.

Type path

`save_freq`

- N>0: save every Nth sample
- N==0: never save a sample
- N==-1: save only the last sample

  Type int

`save_extension`

- .csv: save in textual csv file format
- .ddm: save in binary file format

  Type { ".csv", ".ddm" }

`checkpoint_freq`
Save the state of the session every N seconds.

Type int

`csv_status`
Stores limited set of parameters, indicative for training progress in this file. See `StatusItem`

Type filepath

`addData` (*self*, *pos*, *Y*, *is_scarce=False*, *noise=PyNoiseConfig()*)
Stacks more matrices/tensors next to the main train matrix.

pos [shape] Block position of the data with respect to train. The train matrix/tensor has implicit block position (0, 0).

Y [:class: *numpy.ndarray*, `scipy.sparse` matrix or :class: *SparseTensor*] Data matrix/tensor to add

is_scarce [bool] When *Y* is sparse, and *is_scarce* is *True* the missing values are considered as *unknown*. When *Y* is sparse, and *is_scarce* is *False* the missing values are considered as *zero*. When *Y* is dense, this parameter is ignored.

noise [:class: *PyNoiseConfig*] Noise model to use for *Y*

`addPropagatedPosterior` (*self*, *mode*, *mu*, *Lambda*)
Adds mu and Lambda from propagated posterior

mode [int] dimension to add side info (rows = 0, cols = 1)

mu [:class: *numpy.ndarray* matrix] mean matrix mu should have as many rows as *num_latent* mu should have as many columns as size of dimension *mode* in *train*

Lambda [:class: *numpy.ndarray* matrix] co-variance matrix Lambda should be shaped like K x K x N Where K == *num_latent* and N == dimension *mode* in *train*

`addSideInfo` (*self*, *mode*, *Y*, *noise=PyNoiseConfig()*, *tol=1e-6*, *direct=False*)
Adds fully known side info, for use in with the macau or macauone prior

mode [int] dimension to add side info (rows = 0, cols = 1)

Y [:class: *numpy.ndarray*, `scipy.sparse` matrix] Side info matrix/tensor Y should have as many rows in Y as you have elemnts in the dimension selected using *mode*. Columns in Y are features for each element.

noise [:class: *PyNoiseConfig*] Noise model to use for *Y*

direct [boolean]

- When True, uses a direct inversion method.
- When False, uses a CG solver

The direct method is only feasible for a small (< 100K) number of features.

tol [float] Tolerance for the CG solver.

`addTrainAndTest` (*self*, *Y*, *Ytest=None*, *noise=PyNoiseConfig()*, *is_scarce=True*)
Adds a train and optionally a test matrix as input data to this TrainSession

Parameters

- **`Y`** – Train matrix/tensor
- **`Ytest`** (`scipy.sparse` matrix or :class: *SparseTensor*) – Test matrix/tensor. Mainly used for calculating RMSE.
- **`noise`** – Noise model to use for *Y*
- **`is_scarce`** (*`bool`*) – When *Y* is sparse, and *is_scarce* is *True* the missing values are considered as *unknown*. When *Y* is sparse, and *is_scarce* is *False* the missing values are considered as *zero*. When *Y* is dense, this parameter is ignored.

`getConfig` (*self*)
Get this *TrainSession*'s configuration in ini-file format

`getRmseAvg` (*self*)
Average RMSE across all samples for the test matrix

`getStatus` (*self*)
Returns *`StatusItem`* with current state of the session

`getTestPredictions` (*self*)
Get predictions for test matrix.

Returns list of *`Prediction`*

Return type list

`init` (*self*)
Initializes the *TrainSession* after all data has been added.

You need to call this method befor calling *`step()`*, unless you call *`run()`*

Returns

Return type *`StatusItem`* of the session.

`makePredictSession` (*self*)
Makes a *`PredictSession`* based on the model that as built in this *TrainSession*.

`run` (*self*)
Equivalent to:

```
self.init()
while self.step():
    pass
```

`step` (*self*)
Does on sampling or burnin iteration.

Returns

- **- When a step was executed** (*`StatusItem`* of the session.)

- **- After the last iteration, when no step was executed** (*None*.)

MacauSession

class smurff.**MacauSession**(*Ytrain*, *Ytest=None*, *side_info=None*, *univariate=False*, *direct=False*, ***args*)

A train session specialized for use with the Macau algorithm

Ytrain

Train matrix/tensor

Ytest [`scipy.sparse` matrix or :class: *SparseTensor*] Test matrix/tensor. Mainly used for calculating RMSE.

side_info [list of :class: *numpy.ndarray*, `scipy.sparse` matrix or None] Side info matrix/tensor for each dimension If there is no side info for a certain mode, pass *None*. Each side info should have as many rows as you have elemnts in corresponding dimension of *Ytrain*.

direct [bool] Use Cholesky instead of CG solver

univariate [bool] Use univariate or multivariate sampling.

****args:** Extra arguments are passed to the `TrainSession`

Type

class *numpy.ndarray*, `scipy.sparse` matrix or :class: *SparseTensor*

BPMFSession

class smurff.**BPMFSession**(*Ytrain*, *Ytest=None*, *univariate=False*, ***args*)

A train session specialized for use with the BPMF algorithm

Ytrain

Train matrix/tensor

Ytest [`scipy.sparse` matrix or :class: *SparseTensor*] Test matrix/tensor. Mainly used for calculating RMSE.

univariate [bool] Use univariate or multivariate sampling.

****args:** Extra arguments are passed to the `TrainSession`

Type

class *numpy.ndarray*, `scipy.sparse` matrix or :class: *SparseTensor*

StatusItem

class smurff.**StatusItem**(*phase*, *iter*, *phase_iter*, *model_norms*, *rmse_avg*, *rmse_1sample*, *train_rmse*, *auc_1sample*, *auc_avg*, *elapsed_iter*, *nnz_per_sec*, *samples_per_sec*)

Short set of parameters indicative for the training progress.

phase

Type { “Burnin”, “Sampling” }

`iter`
Current iteration in current phase

Type int

`phase_iter`
Number of iterations in this phase

Type int

`model_norms`
Norm of each U/V matrix

Type list of float

`rmse_avg`
Averag RMSE for test matrix across all samples

Type float

`rmse_1sample`
RMSE for test matrix of last sample

Type float

`train_rmse`
RMSE for train matrix of last sample

Type float

`auc_1sample`
ROC AUC of the test matrix of the last sample Only available if you provided a threshold.

Type float

`auc_avg`
Averag ROC AUC of the test matrix accross all samples Only available if you provided a threshold.

Type float

`elapsed_iter`
Number of seconds the last sampling iteration took

Type float

`nnz_per_sec`
Compute performance indicator; number of non-zero elements in train processed per second

Type float

`samples_per_sec`
Compute performance indicator; number of rows and columns in U/V processed per second

Type float

1.4.3 Inference

Contents

A `PredictSession` provides access to making prediction from saved models. Predictions for a single point are stored in the `Prediction`.

PredictSession

class smurff.**PredictSession**(*root_file*)

Session for making predictions using a model generated using a *TrainSession*.

A *PredictSession* can be made directly from a *TrainSession*

```
>>> predict_session  = train_session.makePredictSession()
```

or from a root file

```
>>> predict_session = PredictSession("root.ini")
```

predict(*coords_or_sideinfo=None*)

Generate predictions on *coords_or_sideinfo*. Parameters specify coordinates of sideinfo/features for each dimension. :param operands: A combination of coordindates in the matrix/tensor and/or features you want to use

to make predictions. *len(coords)* should be equal to number of dimensions in the sample. Each element *coords* can be a:

- int : a single element in this dimension is selected. For example, a single row or column in a matrix.
- `slice` : a slice is selected in this dimension. For example, a number of rows or columns in a matrix.
- None : all elements in this dimension are selected. For example, all rows or columns in a matrix.
- `numpy.ndarray` : 2D numpy array used as dense sideinfo. Each row vector is used as side-info.
- `scipy.sparse.spmatrix` : sparse matrix used as sideinfo. Each row vector is used as side-info.

Returns A `numpy.ndarray` of shape *[N x T1 x T2 x . . .]* where N is the number of samples in this *PredictSession* and *T1 x T2 x . . .* has the same numer of dimensions as the train data.

Return type numpy.ndarray

predict_all()

Computes the full prediction matrix/tensor.

Returns A `numpy.ndarray` of shape *[N x T1 x T2 x . . .]* where N is the number of samples in this *PredictSession* and *T1 x T2 x . . .* is the shape of the train data.

Return type numpy.ndarray

predict_one(*coords_or_sideinfo*, *value=nan*)

Computes prediction for one point in the matrix/tensor

Parameters

- **`coords_or_sideinfo`** (*`tuple of coordinates and/or feature vectors`*) –
- **`value`** (*`float, optional`*) – The *true* value for this point

Returns The prediction

Return type *`Prediction`*

`predict_some`(*test_matrix*)
Computes prediction for all elements in a sparse test matrix

Parameters **`test_matrix`** (*`scipy sparse matrix`*) – Coordinates and true values to make predictions for

Returns list of *`Prediction`* objects.

Return type list

Prediction

`class` `smurff.`**`Prediction`**(*coords*, *val*, *pred_1sample=nan*, *pred_avg=nan*, *var=nan*, *nsamples=-1*)
Stores predictions for a single point in the matrix/tensor

`coords`
Position of this prediction in the train matrix/tensor

Type shape

`val`
True value or "nan" if no true value is known

Type float

`nsamples`
Number of samples collected to make this prediction

Type int

`pred_1sample`
Predicted value using only the last sample

Type float

`pred_avg`
Predicted value using the average prediction across all samples

Type float

`var`
Variance amongst predictions across all samples

Type float

`pred_all`
List of predictions, one for each sample

Type list

`static fromTestMatrix`(*test_matrix*)
Creates a list of predictions from a scipy sparse matrix"

Parameters **`test_matrix`** (*`scipy sparse matrix`*) –

Returns List of *`Prediction`*. Only the coordinate and true value is filled.

Return type list

TODO: FixedNoise, AdaptiveNoise, ProbitNoise

Python Module Index

Index