ProSper Documentation

Release 0.1.0

ProSper Authors

Aug 01, 2019

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This package contains all the source code to reproduce the numerical experiments described in the paper. It contains a parallelized implementation of the Binary Sparse Coding (BSC) [1], Gaussian Sparse Coding (GSC) [2], Maximum Causes Analysis (MCA) [3], Maximum Magnitude Causes Analysis (MMCA) [4], Ternary Sparse Coding (TSC) [5], and Discrete Sparse Coding [7] models. All these probabilistic generative models are trained using a truncated Expectation Maximization (EM) algorithm [6].

1.1 Software dependencies

Python related dependencies can be installed using:

MPI4PY also requires a system level installation of MPI. You can do that on MacOS using Homebrew:

for Ubuntu systems:

for any other system you might wish to review the relevent section of the MPI4PY [installation guidelines](https://mpi4py.readthedocs.io/en/stable/appendix.html#building-mpi)

1.2 Installation

The recommended approch to install the framework is to obtain the most recent stable version from *github.com*:

```
$ git clone git@github.com:mlold/prosper.git
$ cd prosper
$ python setup.py install
```

After installation you should run the testsuite to ensure all neccessary dependencies are installed correctly and that everyting works as expected:

```
$ nosetests -v
```

Optionally you can replace the final line with:

```
$ python setup.py develop
```

This option installs the library using links and it allows the user to edit the library without reinstalling it (useful for Prosper developers).

1.3 Running examples

You are now ready to run your first dictionary learning experiments on artificial data.

Create some artifical training data by running *bars-create-data.py*:

```
$ cd examples/barstests
$ python bars-learnign-and-inference.py param-bars-<...>.py
```

where <…> should be appropriately replaced to correspond to one of the parameter files available in the directory. The bars-run-all.py script should then initialize and run the algorithm which corresponds to the chosen parameter file.

1.3.1 Results/Output

The results produced by the code are stored in a 'results.h5' file under "./output/…/". The file stores the model parameters (e.g., W, pi etc.) for each EM iteration performed. To read the results file, you can use openFile function of the standard tables package in python. Moreover, the results files can also be easily read by other packages such as Matlab etc.

1.4 Running on a parallel architecture

The code uses MPI based parallelization. If you have parallel resources (i.e., a multi-core system or a compute cluster), the provided code can make a use of parallel compute resources by evenly distributing the training data among multiple cores.

To run the same script as above, e.g.,

a) On a multi-core machine with 32 cores:

```
$ mpirun -np 32 bars-learning-and-inference.py param-bars-<...>.py
```

b) On a cluster:

```
$ mpirun --hostfile machines python bars-learning-and-inference.py param-bars-<...>.py
```

where 'machines' contains a list of suitable machines.

See your MPI documentation for the details on how to start MPI parallelized programs.

1.5 References

[1] M. Henniges, G. Puertas, J. Bornschein, J. Eggert, and J. Lücke (2010). Binary Sparse Coding. Proc. LVA/ICA 2010, LNCS 6365, 450-457.

[2] A.-S. Sheikh, J. A. Shelton, J. Lücke (2014). A Truncated EM Approach for Spike-and-Slab Sparse Coding. Journal of Machine Learning Research, 15:2653-2687.

[3] G. Puertas, J. Bornschein, and J. Lücke (2010). The Maximal Causes of Natural Scenes are Edge Filters. Advances in Neural Information Processing Systems 23, 1939-1947.

[4] J. Bornschein, M. Henniges, J. Lücke (2013). Are V1 simple cells optimized for visual occlusions? A comparative study. PLOS Computational Biology 9(6): e1003062.

[5] G. Exarchakis, M. Henniges, J. Eggert, and J. Lücke (2012). Ternary Sparse Coding. International Conference on Latent Variable Analysis and Signal Separation (LVA/ICA), 204-212.

[6] J. Lücke and J. Eggert (2010). Expectation Truncation and the Benefits of Preselection in Training Generative Models. Journal of Machine Learning Research 11:2855-2900.

[7] G. Exarchakis, and J. Lücke (2017). Discrete Sparse Coding. Neural Computation, 29(11), 2979-3013.

CHAPTER

TWO

EXPECTATION MAXIMIZATION INFRASTRUCTURE

The machine learning algorithms distributed with ProSper are based on latent variable probabilistic data models and are trained with variational Expectation Maximization (EM) learning algorithm. The source code is therefore organized under an EM module.

2.1 Expectation Maximization Algorithm

The Expectation Maximization (EM) algorithm is used to optimize probabilistic models with latent random variables. It is an iterative algorithm that optimizes with respect to the posterior distribution in the E-step and and proceeds with an optimization of the model parameters in the M step. A simple implementation is given in the EM class. The more technical components however are contained in the Model classes.

class `prosper.em.`**EM**(*model=None*, *anneal=None*, *data=None*, *lparams=None*, *mpi_comm=None*)

This class drives the EM algorithm.

run(*verbose=False*)

Run a complete cooling-cycle

When *verbose* is True a progress message is printed for every step via `dlog.progress(...)()`

step()

Execute a single EM-Step

class `prosper.em.`**Model**(*comm=<Mock id='140315140465496'>*)

Model Base Class.

Includes knowledge about parameters, data generation, model specific functions, E and M step.

Specific models will be subclasses of this abstract base class.

generate_data(*model_params*, *N*)

Generate datapoints according to the model.

Given the model parameters *model_params* return a dataset of *N* datapoints.

noisify_params(*model_params*, *anneal*)

Noisify model params.

Noisify the given model parameters according to self.noise_policy and the annealing object provided. The noise_policy of some model parameter PARAM will only be applied if the annealing object provides a noise strength via PARAM_noise.

standard_init(*data*)

Initialize a set of model parameters in some sane way.

Return value is model_parameter dictionary

step (*anneal*, *model_params*, *my_data*)

2.2 Component Analysis Models

Component Analysis Models refers to models with multiple latent variables may contribute to the same datapoints to

class prosper.em.camodels.**CAModel** (*D, H, Hprime, gamma, to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140315115818624'>*)

Abstract base class for Sparse Coding models with binary latent variables and expectation tuncation (ET) based training scheme.

This

check_params (*model_params*)

Perform a sanity check on the model parameters. Throw an exception if there are major violations; correct the parameter in case of minor violations

compute_lpj (*anneal*, *model_params*, *my_data*)

Determine candidates and compute log-pseudo-joint.

Parameters

- **anneal** (*prosper.em.annealling.Annealing*) – Annealing schedule, e.g., em.anneal
- **model_params** (*dict*) – Learned model parameters, e.g., em.lparams
- **my_data** (*dict*) – Data stored in field 'y'.

generate_data (*model_params*, *my_N*)

Generate data according to the model. Internally uses generate_data_from_hidden.

Parameters

- **model_params** (*dict*) – Ground-truth model parameters to use
- **my_N** (*int*) – number of datapoints to generate on this MPI rank

This method does _not_ obey gamma: The generated data may have more than gamma active causes for a given datapoint.

inference (*anneal*, *model_params*, *test_data*, *topK=10*, *logprob=False*, *adaptive=True*, *Hprime_max=None*, *gamma_max=None*)

Perform inference with the learned model on test data and return the top K configurations with their posterior probabilities. :param anneal: Annealing schedule, e.g., em.anneal :type anneal: prosper.em.annealling.Annealing :param model_params: Learned model parameters, e.g., em.lparams :type model_params: dict :param test_data: The test data stored in field 'y'. Candidates stored in 'candidates' (optional). :type test_data: dict :param topK: The number of returned configurations :type topK: int :param logprob: Return probability or log probability :type logprob: boolean :param adaptive: Adjust Hprime, gamma to be greater than the number of active units in the MAP state :type adaptive: boolean :param Hprime_max: Upper limit for Hprime adjustment :type Hprime_max: int :param gamma_max: Upper limit for gamma adjustment :type gamma_max: int

select_partial_data (*anneal*, *my_data*)

Select a partial data-set from my_data and return it.

The fraction of datapoints selected is determined by anneal['partial']. If anneal['partial'] is equal to either 1 or 0 the whole dataset will be returned.

standard_init (*data*)

Standard onitial estimation for model parameters.

This implementation

W and *sigma*.

each *W* raw is set to the average over the data plus WGN of mean zero and var *sigma*/4. *sigma* is set to the variance of the data around the computed mean. *pi* is set to 1./H . Returns a dict with the estimated parameter set with entries "W", "pi" and "sigma".

step (*anneal*, *model_params*, *my_data*)
Perform an EM-step

prosper.em.camodels.**generate_state_matrix** (*Hprime*, *gamma*)
Full combinatorics of Hprime-dim binary vectors with at most gamma ones.

Parameters

- **Hprime** (*int*) – Vector length
- **gamma** – Maximum number of ones
- **gamma** – int

2.2.1 Binary Sparse Coding

class prosper.em.camodels.bsc_et.**BSC_ET** (*D, H, Hprime, gamma, to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140315333505656'>*)
Binary Sparse Coding

Implements learning and inference of a Binary Sparse coding model under a variational approximation

comm

Type MPI communicator

D
number of features

Type int

gamma
approximation parameter for maximum number of non-zero states

Type int

H
number of latent variables

Type int

Hprime
approximation parameter for latent space trunctation

Type int

K
number of different values the latent variables can take

Type int

no_states
number of different states of latent variables except singleton states and zero state

Type (.., Hprime) ndarray

single_state_matrix
matrix that holds all possible singleton states

Type ((K-1)*H, H) ndarray

state_abs
number of non-zero elements in the rows of the state_matrix

Type (no_states,) ndarray

state_matrix
latent variable states taken into account during the em algorithm

Type (no_states, Hprime) ndarray

states
the differnt values that a latent variable can take must include 0 and one more integer

Type (K,) ndarray

to_learn
list of strings included in model_params.keys() that specify which parameters are going to be optimized

Type list

References

[1] M. Henniges, G. Puertas, J. Bornschein, J. Eggert, and J. Lücke (2010). Binary Sparse Coding. Proc. LVA/ICA 2010, LNCS 6365, 450-457.

[2] J. Lücke and J. Eggert (2010). Expectation Truncation and the Benefits of Preselection in Training Generative Models. Journal of Machine Learning Research 11:2855-2900.

E_step (*anneal*, *model_params*, *my_data*)
BSC E_step

my_data variables used:

my_data['y'] Datapoints my_data['can'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

M_step (*anneal*, *model_params*, *my_suff_stat*, *my_data*)
BSC M_step

my_data variables used:

my_data['y'] Datapoints my_data['candidates'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

generate_from_hidden (*model_params*, *my_hdata*)
Generate data according to the MCA model while the latents are given in my_hdata['s'].

This method does _not_ obey gamma: The generated data may have more than gamma active causes for a given datapoint.

select_Hprimes(*model_params, data*)

Return a new data-dictionary which has been annotated with a data['candidates'] dataset. A set of self.Hprime candidates will be selected.

2.2.2 Ternary Sparse Coding

class prosper.em.camodels.tsc_et.**TSC_ET**(*D, H, Hprime, gamma, to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140315053104040'>*)

Ternary Sparse Coding

Implements learning and inference of a Ternary Sparse coding model under a variational approximation

comm

Type MPI communicator

D

number of features

Type int

gamma

approximation parameter for maximum number of non-zero states

Type int

H

number of latent variables

Type int

Hprime

approximation parameter for latent space trunctation

Type int

K

number of different values the latent variables can take

Type int

no_states

number of different states of latent variables except singleton states and zero state

Type (.., Hprime) ndarray

single_state_matrix

matrix that holds all possible singleton states

Type ((K-1)*H, H) ndarray

state_abs

number of non-zero elements in the rows of the state_matrix

Type (no_states,) ndarray

state_matrix

latent variable states taken into account during the em algorithm

Type (no_states, Hprime) ndarray

states

the differnt values that a latent variable can take must include 0 and one more integer

Type (K,) ndarray

to_learn
list of strings included in model_params.keys() that specify which parameters are going to be optimized

Type list

References

[1] G. Exarchakis, M. Henniges, J. Eggert, and J. Lücke (2012). Ternary Sparse Coding. International Conference on Latent Variable Analysis and Signal Separation (LVA/ICA), 204-212.

[2] J. Lücke and J. Eggert (2010). Expectation Truncation and the Benefits of Preselection in Training Generative Models. Journal of Machine Learning Research 11:2855-2900.

E_step (*anneal*, *model_params*, *my_data*)
E step for Teranary Sparse Coding Identifies approximate posterior information for Ternary Sparse Coding

Parameters

- **anneal** (*Annealing object*) –

  contains information related to annealing

  anneal['T']: scalar Temperature for det. annealing

  anneal['N_cut_factor']: scalar 0.: no truncation; 1. trunc. according to model

- **model_params** (*dict*) –

  dictionary of parameters

  model_params['W']: ndarray dictionary

  model_params['sigma']: float standard deviation of gaussian noise

  model_params['pi']: float prior parameter

- **my_data** (*dict*) –

  datapoints dictionary

  my_data['y']: ndarray Datapoints

  my_data['can']: ndarray Candidate H's according to selection func.

Returns

dict['logpj'] Approximate joint of datapoints and latent variable states

Return type dict

M_step (*anneal*, *model_params*, *my_suff_stat*, *my_data*)
Ternary Sparse Coding M-Step

This function is responsible for finding the optimal model parameters given an approximation of the posterior distribution.

Parameters

- **anneal** (*Annealing object*) –

  Annealing type obje ct containing training schedule information anneal['T'] : Temperature for det. annealing anneal['N_cut_factor']: 0. no truncation; 1. trunc. according to model

- **model_params** (*dict*) –

dictionary containing model parameters

model_params['W']: (H,D) ndarray linear dictionary

model_params['pi']: (K,) ndarray prior parameters

model_params['sigma']: float standard deviation of noise model

- **my_suff_stat** (*dict*) –

  dictionary containing inforamtion about the joint distribution

  my_suff_stat['logpj']: (my_N,no_states) ndarray logarithm of joint of data and latent variable states

- **my_data** (*dict*) –

  data dictionary

  my_data['y']: (my_N,D) ndarray datapoints

  my_data['candidates']: (my_n,Hprime) Candidate H's according to selection func.

Returns

dictionary containing updated model parameters

dict['W']: (H,D) ndarray linear dictionary

dict['pi']: (K,) ndarray prior parameters

dict['sigma']: float standard deviation of noise model

Return type dict

generate_data (*model_params*, *my_N*)

Parameters

- **model_params** (*dict*) –

  model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **my_N** (*int*) – number of datapoints for this process

Returns

- *dict* –

  returns generated data

  dict['y']: (my_N, D) ndarray generated data

  dict['s']: (my_N, H) ndarray latent variable states that generated the data

- *Deleted Parameters*
- ———————
- **noise_on** (*bool, optional*) – flag to control deterministic/stochastic generation. If True gaussian noise with standard deviation model_params['sigma'] is added to the data

- **gs** (*(my_N, H), optional*) – ground truth latent variables. This option is used for generating artificial data with particular latent variables. Defaults to randomly sampled latent variables from the prior
- **gp** (*(my_N, H), optional*) – ground truth posterior. This option is used for generating data that have a particular true posterior distribution. Defaults to randomly sampled latent variables from the prior

inference(*anneal*, *model_params*, *test_data*, *topK=10*, *logprob=False*, *abs_marginal=True*, *adaptive=True*, *Hprime_max=None*, *gamma_max=None*)

Perform inference with the learned model on test data and return the top K configurations with their posterior probabilities.

Parameters

- **anneal** (*Annealing object*) – annealing information
- **model_params** (*dict*) – dictionary with model parameters
- **test_data** (*dict*) – data dictionary. The data in this case are ndarray under the key 'y'.
- **topK** (*int, optional*) – the number of most probable latent variable states to be returned
- **logprob** (*bool, optional*) – the probabilities of the most probable latent variable states
- **abs_marginal** (*bool, optional*) – Description
- **adaptive** (*bool, optional*) – if set to True it will run inference again for datapoints with gamma active latent variables in the top state using setting gamma=gamma+1 and Hprime=Hprime+1
- **Hprime_max** (*None, optional*) – if adaptive is True it will stop Hprime from increasing above this integer. None defaults to H.
- **gamma_max** (*None, optional*) – if adaptive is True it will stop gamma from increasing above this integer. None defaults to H.

Returns

a dictionary with posterior information

dict['s']: (batchsize, topK, H) ndarray the topK most probable vectors

dict['m']: (batchsize, H) ndarray latent variable marginal distribution

dict['am']: (batchsize, H) ndarray absolote latent variable marginal distribution

dict['p']: (batchsize, topK) ndarray probabilities of topK latent variable states

dict['gamma']: int sparseness approximation parameter

dict['Hprime']: int truncation approximation parameter

Return type dict

select_Hprimes(*model_params*, *data*)

Return a new data-dictionary which has been annotated with a data['candidates'] dataset. A set of self.Hprime candidates will be selected.

Parameters

- **model_params** (*dict*) –

  dictionary containing model parameters

model_params['W']: (H,D) ndarray linear dictionary

model_params['pi']: (K,) ndarray prior parameters

model_params['sigma']: float standard deviation of noise model

- **data** (*dict*) –

  dataset dictionary

  data['y']: (my_n,D) ndarray datapoints

Returns

dataset dictionary

data['y']: (my_n,D) ndarray datapoints

data['candidates']: (my_n,) ndarray indices of the best explained datapoints

Return type dict

2.2.3 Discrete Sparse Coding

class prosper.em.camodels.dsc_et.**DSC_ET**(*D, H, Hprime, gamma, states=array([-1., 0., 1.]), to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140315051498856'>*)

Discrete Sparse Coding

Implements learning and inference of a Discrete Sparse coding model under a variational approximation

comm

Type MPI communicator

D

number of features

Type int

gamma

approximation parameter for maximum number of non-zero states

Type int

H

number of latent variables

Type int

Hprime

approximation parameter for latent space trunctation

Type int

K

number of different values the latent variables can take

Type int

no_states

number of different states of latent variables except singleton states and zero state

Type (.., Hprime) ndarray

single_state_matrix
matrix that holds all possible singleton states

Type ((K-1)*H, H) ndarray

state_abs
number of non-zero elements in the rows of the state_matrix

Type (no_states,) ndarray

state_matrix
latent variable states taken into account during the em algorithm

Type (no_states, Hprime) ndarray

states
the differnt values that a latent variable can take must include 0 and one more integer

Type (K,) ndarray

to_learn
list of strings included in model_params.keys() that specify which parameters are going to be optimized

Type list

References

[1] G. Exarchakis, and J. Lücke (2017). Discrete Sparse Coding. Neural Computation, 29(11), 2979-3013.

[2] J. Lücke and J. Eggert (2010). Expectation Truncation and the Benefits of Preselection in Training Generative Models. Journal of Machine Learning Research 11:2855-2900.

E_step (*anneal*, *model_params*, *my_data*)
Discrete Sparse Coding E-step

This function is responsible for finding an approximation of the posterior distribution given the model parameters.

Parameters

- **anneal** (*Annealing object*) –

  Annealing type obje ct containing training schedule information anneal['T'] : Temperature for det. annealing anneal['N_cut_factor']: 0. no truncation; 1. trunc. according to model

- **model_params** (*dict*) –

  dictionary containing model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **my_data** (*dict*) –

  data dictionary

  my_data['y']: (my_N,D) ndarray datapoints

  my_data['candidates']: (my_n,Hprime) Candidate H's according to selection func.

Returns

returns information about the approximation posterior

dict['logpj']: (my_n,no_states) ndarray an approximation of the logarithm of the joint distribution

Return type dict

`M_step` (*anneal*, *model_params*, *my_suff_stat*, *my_data*)
Discrete Sparse Coding M-Step

This function is responsible for finding the optimal model parameters given an approximation of the posterior distribution.

Parameters

- **`anneal`** (*`Annealing object`*) –

  Annealing type obje ct containing training schedule information anneal['T'] : Temperature for det. annealing anneal['N_cut_factor']: 0. no truncation; 1. trunc. according to model

- **`model_params`** (*`dict`*) –

  dictionary containing model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **`my_suff_stat`** (*`dict`*) –

  dictionary containing inforamtion about the joint distribution

  my_suff_stat['logpj']: (my_N,no_states) ndarray logarithm of joint of data and latent variable states

- **`my_data`** (*`dict`*) –

  data dictionary

  my_data['y']: (my_N,D) ndarray datapoints

  my_data['candidates']: (my_n,Hprime) Candidate H's according to selection func.

Returns

dictionary containing updated model parameters

dict['W']: (H,D) ndarray linear dictionary

dict['pi']: (K,) ndarray prior parameters

dict['sigma']: float standard deviation of noise model

Return type dict

`check_params` (*model_params*)
Sanity check.

Sanity-check the given model parameters. Raises an exception if something is severely wrong.

Parameters **`model_params`** (*`dict`*) –

dictionary of model parameters

model_params['W']: (H,D) ndarray linear dictionary

model_params['pi']: (K,) ndarray prior parameters

model_params['sigma']: float standard deviation of noise model

Returns

model parameters

model_params['W']: (H,D) ndarray linear dictionary

model_params['pi']: (K,) ndarray prior parameters

model_params['sigma']: float standard deviation of noise model

Return type dict

`free_energy`(*model_params*, *my_data*)
Deprecated

`gain`(*old_parameters*, *new_parameters*)
Deprecated

`generate_data`(*model_params*, *my_N*, *noise_on=True*, *gs=None*, *gp=None*)

Parameters

- **`model_params`** (*`dict`*) –

  model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **`my_N`** (*`int`*) – number of datapoints for this process
- **`noise_on`** (*`bool, optional`*) – flag to control deterministic/stochastic generation. If True gaussian noise with standard deviation model_params['sigma'] is added to the data
- **`gs`** (*`(my_N, H), optional`*) – ground truth latent variables. This option is used for generating artificial data with particular latent variables. Defaults to randomly sampled latent variables from the prior
- **`gp`** (*`(my_N, H), optional`*) – ground truth posterior. This option is used for generating data that have a particular true posterior distribution. Defaults to randomly sampled latent variables from the prior

Returns

returns generated data

dict['y']: (my_N, D) ndarray generated data

dict['s']: (my_N, H) ndarray latent variable states that generated the data

Return type dict

`get_likelihood`(*D*, *sigma*, *logpj_all*, *N*)
Data likelihood This functions computes the approximate likelihood of the data from the approximation of the joint

Parameters

- **`D`** (*`int`*) – Number of observed dimensions

- **sigma** (*float*) – standard deviation of the noise model
- **logpj_all** (*(my_N,no_states) ndarray*) – approximation of the joint probability of the data and the latent variable states
- **N** (*int*) – total number of datapoints. Useful in parallel execution

Returns the approximate likelihood value

Return type float

inference(*anneal*, *model_params*, *test_data*, *topK=10*, *logprob=False*, *adaptive=True*, *Hprime_max=None*, *gamma_max=None*)

Perform inference with the learned model on test data and return the top K configurations with their posterior probabilities.

Parameters

- **anneal** (*Annealing object*) – annealing information
- **model_params** (*dict*) – dictionary with model parameters
- **test_data** (*dict*) – data dictionary. The data in this case are ndarray under the key 'y'.
- **topK** (*int, optional*) – the number of most probable latent variable states to be returned
- **logprob** (*bool, optional*) – the probabilities of the most probable latent variable states
- **adaptive** (*bool, optional*) – if set to True it will run inference again for datapoints with gamma active latent variables in the top state using setting gamma=gamma+1 and Hprime=Hprime+1
- **Hprime_max** (*None, optional*) – if adaptive is True it will stop Hprime from increasing above this integer. None defaults to H.
- **gamma_max** (*None, optional*) – if adaptive is True it will stop gamma from increasing above this integer. None defaults to H.

Returns a dictionary with posterior information

Return type dict

noisify_params(*model_params*, *anneal*)

Noisify model params.

Noisify the given model parameters according to self._noise_policy and the annealing object provided. The noise_policy of some model parameter PARAM will only be applied if the annealing object provides a noise strength via PARAM_noise.

Parameters

- **model_params** (*dict*) –

  dictionary containing model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **anneal** (*Annealing object*) –

Annealing type object containing training schedule information
anneal['W_noise'] : standard deviation of noise to be added to the dictionary anneal['pi_noise']: standard deviation of noise to be added to the prior anneal['sigma_noise']: standard deviation of noise to be added to the standard deviation of the noise model

Returns

model_params [dict]

dictionary containing model parameters

model_params['W']: (H,D) ndarray linear dictionary

model_params['pi']: (K,) ndarray prior parameters

model_params['sigma']: float standard deviation of noise model

Return type dict

`select_Hprimes`(*model_params*, *data*)

Return a new data-dictionary which has been annotated with a data['candidates'] dataset. A set of self.Hprime candidates will be selected.

Parameters

- **`model_params`** (*`dict`*) –

  model parameters

  model_params['W']: (H,D) ndarray linear dictionary

  model_params['pi']: (K,) ndarray prior parameters

  model_params['sigma']: float standard deviation of noise model

- **`data`** (*`dict`*) –

  dataset dictionary

  data['y']: (my_n,D) ndarray datapoints

Returns

dataset dictionary

data['y']: (my_n,D) ndarray datapoints

data['candidates']: (my_n,) ndarray indices of the best explained datapoints

Return type dict

`select_partial_data`(*anneal*, *my_data*)

Select a partial data-set from my_data and return it.

The fraction of datapoints selected is determined by anneal['partial']. If anneal['partial'] is equal to either 1 or 0 the whole dataset will be returned.

Parameters

- **`anneal`** (*`Annealing object`*) –

  Annealing type object containing training schedule information
  anneal['partial'] : fraction of the data to return

- **`my_data`** (*`dict`*) –

  dictionary of the dataset

my_data['y']: (my_N, D) ndarray the datapoints

Returns

dictionary of the dataset

my_data['y']: (my_N*anneal['partial'], D) ndarray The updated datapoints

Return type dict

standard_init(*data*)

Standard onitial estimation for model parameters.

This implementation each "W" raw is set to the average over the data plus white Gaussian noise of mean zero and standard deviation "sigma"/4. sigma is set to the variance of the data around the computed mean. "pi" is set to 1./H . Returns a dict with the estimated parameter set with entries "W", "pi" and "sigma".

Parameters **data** (*dict*) – dataset dictionary. Contains a ndarray of size number of samples x number of features under data['y']

Returns a dictionary containing the model parameters

Return type dict

2.2.4 Spike-and-Slab Sparse Coding

class prosper.em.camodels.gsc_et.**GSC**(*D, H, Hprime=0, gamma=0, sigma_sq_type='scalar', to_learn=['W', 'pi', 'mu', 'sigma_sq', 'psi_sq'], comm=<Mock id='140315050039560'>*)

check_params(*model_params*)

Sanity check.

Sanity-check the given model parameters. Raises an exception if something is severely wrong.

compute_lpj(*anneal*, *model_params*, *my_data*)

Determine candidates and compute log-pseudo-joint.

Parameters

- **anneal** (*prosper.em.annealling.Annealing*) – Annealing schedule, e.g., em.anneal
- **model_params** (*dict*) – Learned model parameters, e.g., em.lparams
- **my_data** (*dict*) – Data stored in field 'y'. Candidates stored in 'candidates' (optional).

generate_data(*model_params*, *my_N*)

given ground truth model parameters, generate data of size my_N

generate_from_hidden(*model_params*, *my_hdata*)

Generate data according to the MCA model while the latents are given in my_hdata['s'].

This method does _not_ obey gamma: The generated data may have more than gamma active causes for a given datapoint.

resume_init(*h5_result_file*)

Initialize model parameters to previously inferred values.

standard_init(*my_data*)

Standard Initial of the model parameters.

2.2.5 Maximum Component Analysis

class prosper.em.camodels.mca_et.**MCA_ET**(*D, H, Hprime, gamma, to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140314963439456'>*)

E_step(*anneal, model_params, my_data*)
MCA E_step

my_data variables used:

my_data['y'] Datapoints my_data['can'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing AND softmax anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

M_step(*anneal, model_params, my_suff_stat, my_data*)
MCA M_step

my_data variables used:

my_data['y'] Datapoints my_data['candidates'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing AND softmax anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

check_params(*model_params*)
Sanity-check the given model parameters. Raises an exception if something is severely wrong.

generate_data(*model_params, my_N*)
Generate data according to the MCA model.

This method does _not_ obey gamma: The generated data may have more than gamma active causes for a given datapoint.

select_Hprimes(*model_params, data*)
Return a new data-dictionary which has been annotated with a data['candidates'] dataset. A set of self.Hprime candidates will be selected.

2.2.6 Maximum Magnitude Component Analysis

class prosper.em.camodels.mmca_et.**MMCA_ET**(*D, H, Hprime, gamma, to_learn=['W', 'pi', 'sigma'], comm=<Mock id='140314963998312'>*)

E_step(*anneal, model_params, my_data*)
MCA E_step

my_data variables used:

my_data['y'] Datapoints my_data['can'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing AND softmax anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

M_step (*anneal*, *model_params*, *my_suff_stat*, *my_data*)
MCA M_step

my_data variables used:

my_data['y'] Datapoints my_data['candidates'] Candidate H's according to selection func.

Annealing variables used:

anneal['T'] Temperature for det. annealing AND softmax anneal['N_cut_factor'] 0.: no truncation; 1. trunc. according to model

check_params (*model_params*)
Sanity-check the given model parameters. Raises an exception if something is severely wrong.

generate_from_hidden (*model_params*, *my_hdata*)
Generate data according to the MCA model while the latents are given in my_hdata['s'].

select_Hprimes (*model_params*, *data*)
Return a new data-dictionary which has been annotated with a data['candidates'] dataset. A set of self.Hprime candidates will be selected.

2.3 Mixture Models

Mixture Models refers to models where a single latent variable is responsible for the generation of a datapoint

class prosper.em.mixturemodels.**MixtureModel** (*D, H, to_learn=['W', 'pies'], comm=<Mock id='140314963086528'>*)

check_params (*model_params*)
Sanity check.

Sanity-check the given model parameters. Raises an exception if something is severely wrong.

generate_data (*model_params*, *my_N*)
Generate data according to the model. Internally uses generate_data_from_hidden.

This method does _not_ obey gamma: The generated data may have more than gamma active causes for a given datapoint.

inference (*anneal*, *model_params*, *my_data*, *no_maps=10*)
To be implemented

select_partial_data (*anneal*, *data*)
Select a partial data-set from data and return it.

The fraction of datapoints selected is determined by anneal['partial']. If anneal['partial'] is equal to either 1 or 0 the whole dataset will be returned.

standard_init (*data*)
Standard Initial Estimation for *W* and *sigma*.

each *W* raw is set to the average over the data plus WGN of mean zero and var *sigma*/4. *sigma* is set to the variance of the data around the computed mean. *pi* is set to 1./H . Returns a dict with the estimated parameter set with entries "W", "pi" and "sigma".

step (*anneal*, *model_params*, *data*)
Perform an EM-step

2.3.1 Mixture of Gaussians

Standard mixture of Gaussians model:

class prosper.em.mixturemodels.MoG.**MoG**(*D, H, to_learn=['pies', 'W', 'sigmas_sq'], sigmas_sq_type='full', comm=<Mock id='140314962699096'>*)

check_params(*model_params*)
Sanity check.

Sanity-check the given model parameters. Raises an exception if something is severely wrong.

generate_from_hidden(*model_params*, *my_hdata*)
Generate datapoints according to the model.

Given the model parameters *model_params* return a dataset of *N* datapoints.

resume_init(*h5_output*)
Standard Initial Estimation for W, sigma and mu.

standard_init(*my_data*)
Standard Initial Estimation for W, sigmas and pies.

2.3.2 Mixture of Gaussians

Standard mixture model with a Poisson observation noise model:

class prosper.em.mixturemodels.MoP.**MoP**(*D, H, to_learn=['pies', 'W'], A=nan, comm=<Mock id='140314963205592'>*)

check_params(*model_params*)
Sanity check.

Sanity-check the given model parameters. Raises an exception if something is severely wrong.

generate_from_hidden(*model_params*, *my_hdata*)
Generate datapoints according to the model.

Given the model parameters *model_params* return a dataset of *N* datapoints.

resume_init(*h5_output*)
Standard Initial Estimation for W, sigma and mu.

standard_init(*my_data*)
Standard Initial Estimation for W, sigma and mu.

2.4 Annealing

The annealing module holds utilities relevant to making minor modifications to the training process.

2.4.1 Annealing Class

This is a generic class inheritted by all annealing objects:

class prosper.em.annealing.**Annealing**

Base class for implementations of annealing schemes.

Implementations deriving from this class control the cooling schedule and provide some additional control functions used in the EM algorithm.

next(*gain*)

Returns a (accept, T, finished)-tuple.

***accept* is a boolean and indicates if the parameters changed by *gain* last iteration,** EM should accept the new parameters or if it should bae the next iteration on the old ones.

***finished* is also a boolean and indicate whether the cooling has finished and EM should** drop out of the loop.

T is the temperature EM should use in the next iteration

reset()

Reset the cooling-cycle. This call returs the initial cooling temperature that will be used for the first step.

2.4.2 Linear Annealing

The linear annealing class is an example annealing class that makes changes at hyperparameters changing with linear rate over EM iterations.

class prosper.em.annealing.**LinearAnnealing**(*steps=80*)

as_dict()

Return all annealing parameters with their current value as dict.

next(*gain=0.0*)

Step forward by one step.

After calling this method, this annealing object will potentially return different values for all its values.

reset()

Reset the cooling-cycle. This call returs the initial cooling temperature that will be used for the first step.

CHAPTER

THREE

PROSPER TUTORIAL

This tutorial is one of our top priorities. Unfortunately, we are only able to provide examples at this point. The following example is a simple script that runs a Binary Sparse Coding model on an artificial dataset. It is ready for use in a parallel programming environment with data logging capabilities:

```
#!/usr/bin/env python
import sys

import numpy as np
from mpi4py import MPI

from prosper.utils import create_output_path
from prosper.utils.parallel import pprint, stride_data
from prosper.utils.barstest import generate_bars_dict

from prosper.utils.datalog import dlog, StoreToH5, TextPrinter, StoreToTxt

from prosper.em import EM
from prosper.em.annealing import LinearAnnealing
from prosper.em.camodels.bsc_et import BSC_ET


#=============================================================================
# Parameters

D2     = 5
N      = 1000
Hprime = 6
gamma  = 5

#=============================================================================
# Main
comm = MPI.COMM_WORLD

pprint("="*70)
pprint(" Running %d parallel processes" % comm.size)
pprint("="*70)

H = 2*D2      # number of latent units
D = D2**2     # total size of image in pixels

my_N = N // comm.size

# Some sanity checks
assert Hprime <= H
```

(continues on next page)

(continued from previous page)

```
assert gamma <= Hprime
assert D == D2**2

# Configure DataLogger
print_list = ('T', 'pi', 'sigma')
dlog.set_handler(print_list, TextPrinter)

# Invent some ground truth parameter models
params_gt = {
    'W'     :  10*generate_bars_dict(H),
    'pi'    :  2. / H,
    'sigma' :  1.0
}

# Use model to generate data
model = BSC_ET(D, H, Hprime, gamma)
my_data = model.generate_data(params_gt, my_N)

model_params = model.standard_init(my_data)

# Choose annealing schedule
anneal = LinearAnnealing(50)
anneal['T'] = [(15, 1.), (-10, 1.)]
anneal['Ncut_factor'] = [(0,0.),(2./3,1.)]
anneal['anneal_prior'] = False

# Create and start EM annealing
em = EM(model=model, anneal=anneal)
em.data = my_data
em.lparams = model_params
em.run()

dlog.close()
pprint("Done")
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

H

I

K

L

M

N

P

R

S

T